Exhibit 1

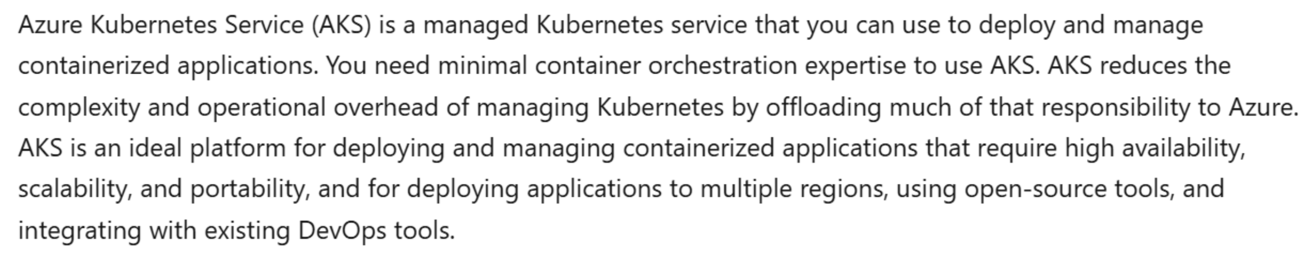
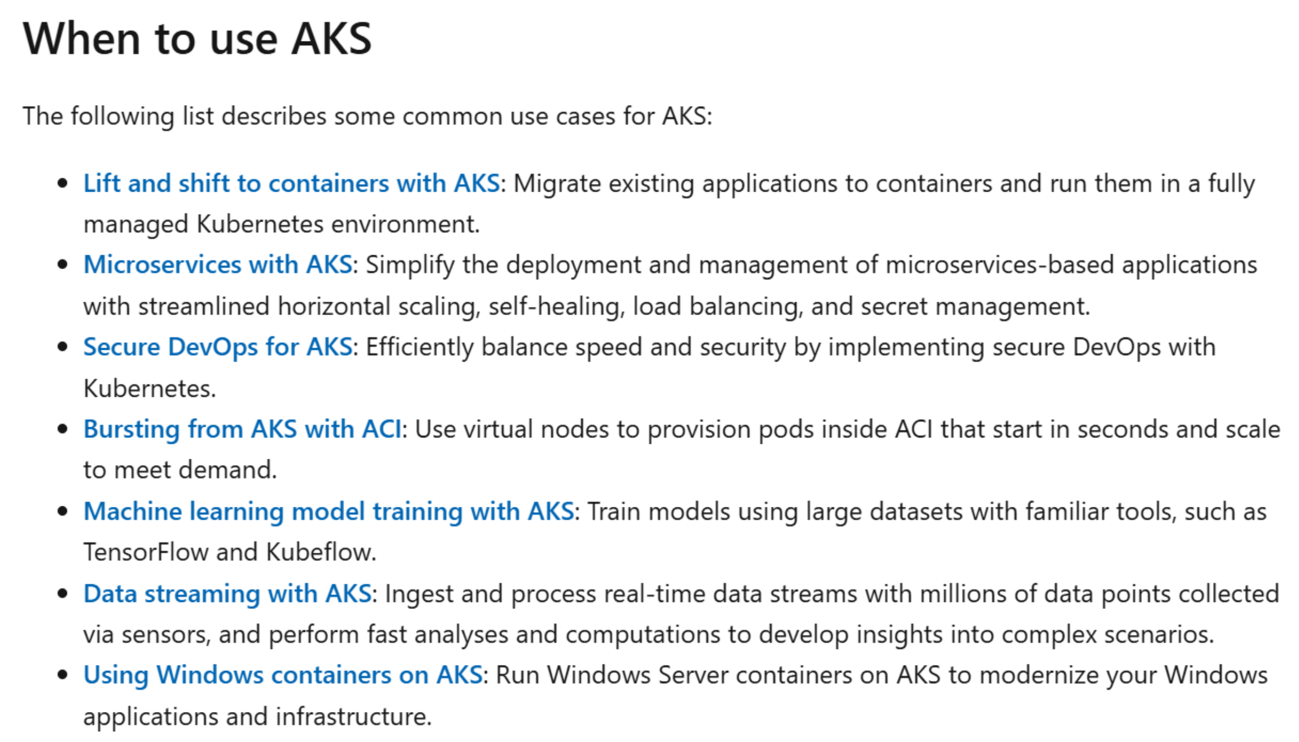
Exhibit 2

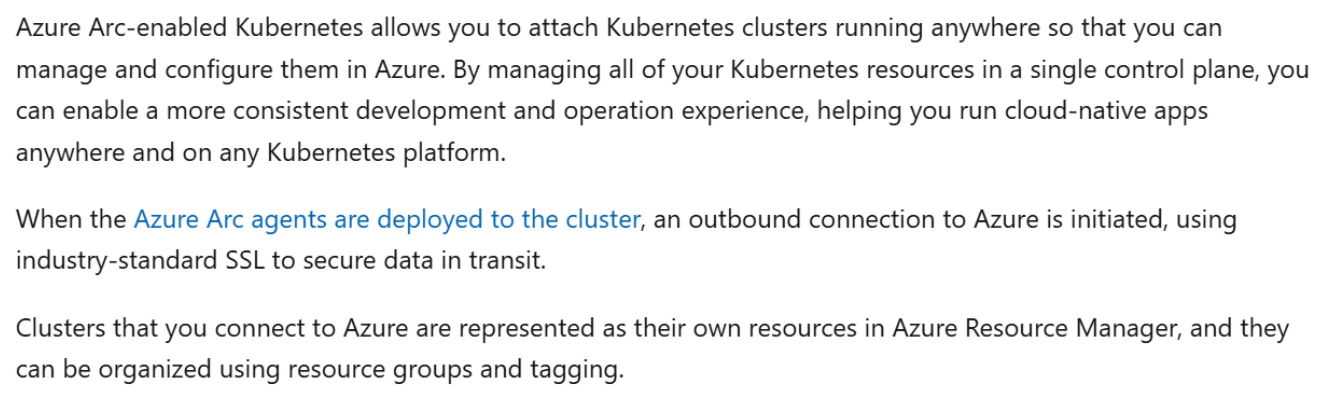
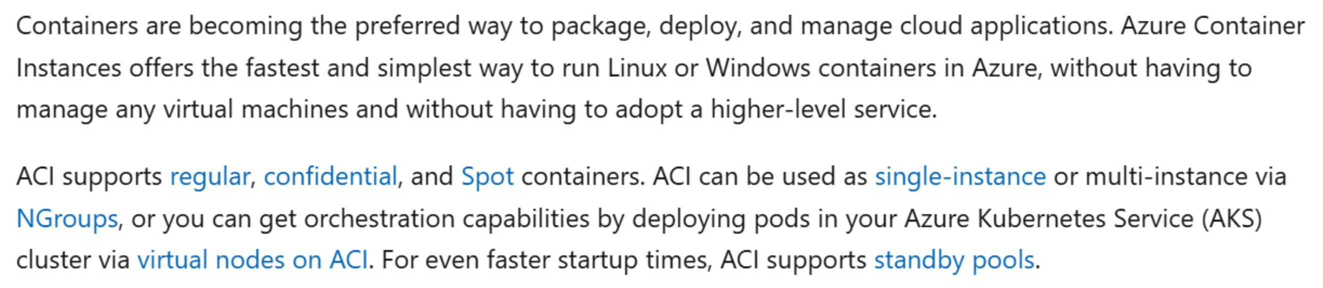
U.S. Patent No. 7,519,814 vs. Microsoft

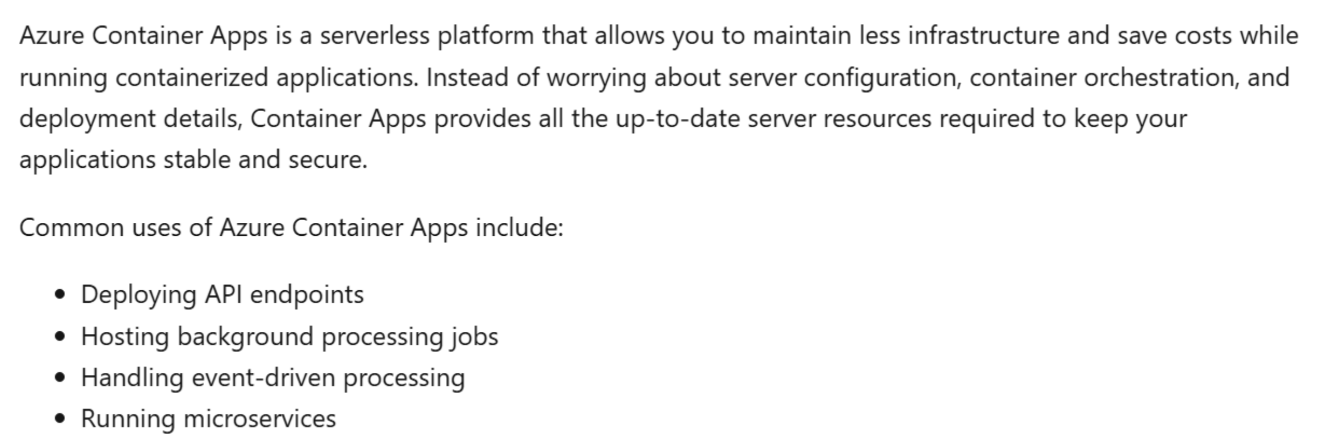
Accused Instrumentalities: Microsoft products and services using secure containerized applications, including without limitation Azure Kubernetes Service ("AKS"), Azure Arc-enabled Kubernetes, Azure Container Registry, and Azure Container Apps, and all versions and variations thereof since the issuance of the asserted patent.

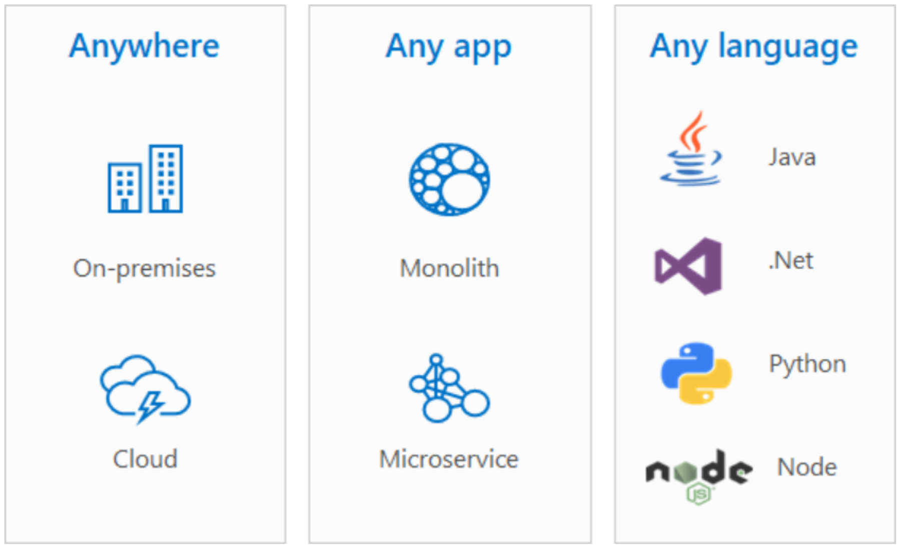
Claim 1

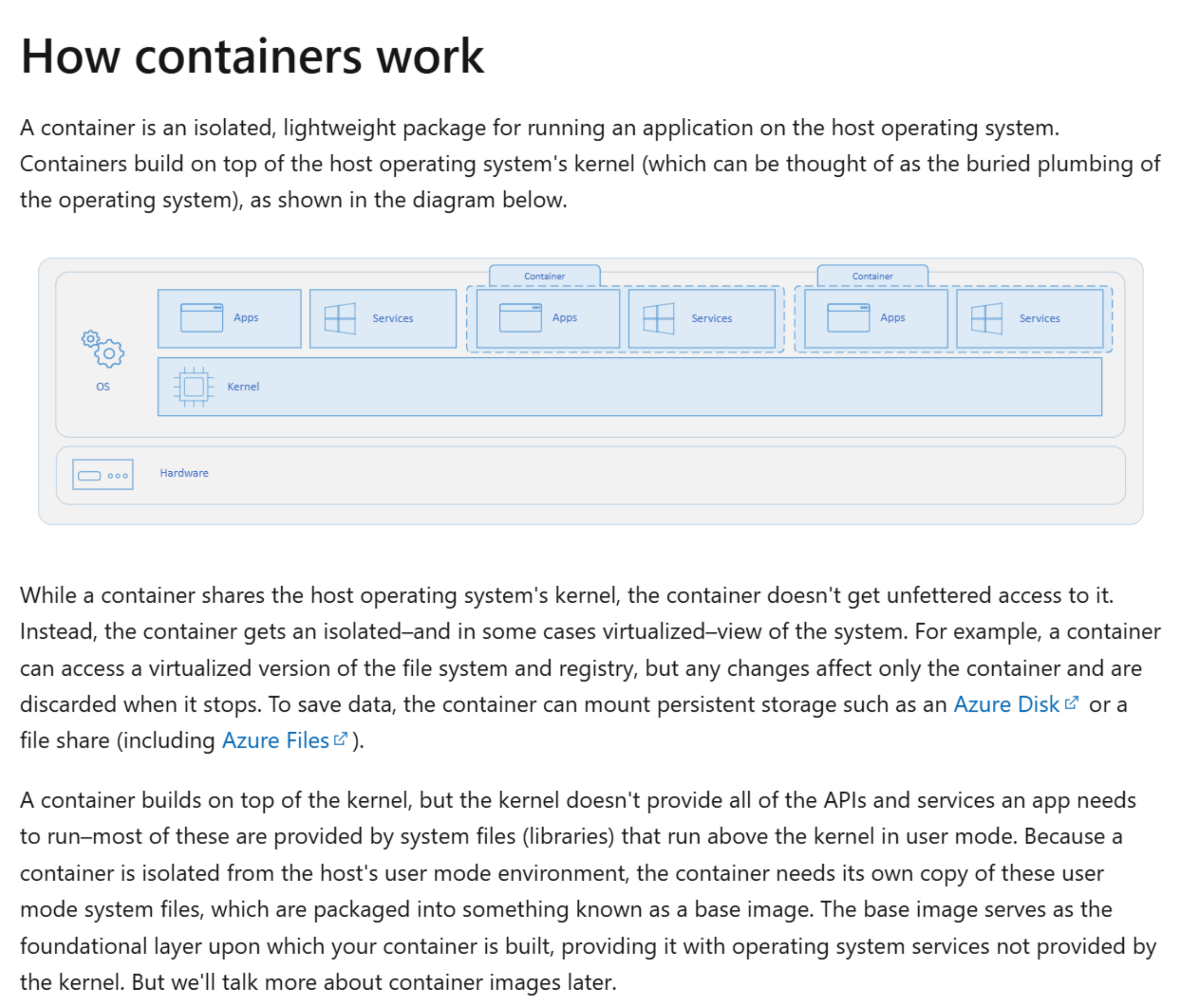
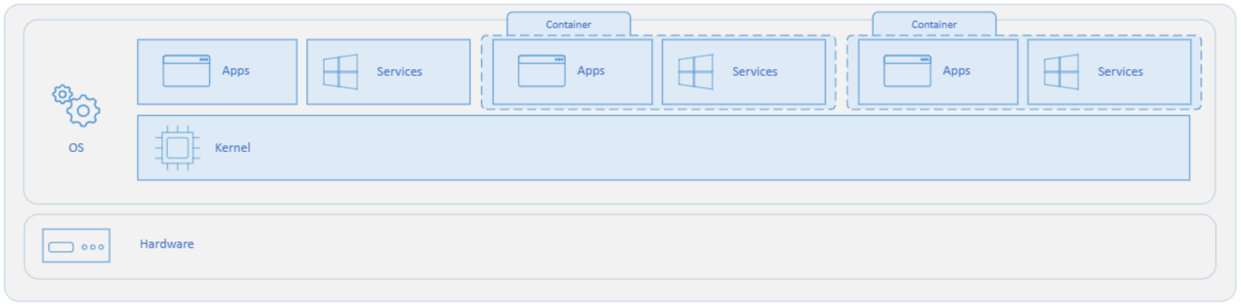
Claim 1	Accused Instrumentalities
[1pre] 1. In a system having a plurality of servers with operating systems that differ, operating in disparate computing environments, wherein each server includes a processor and an operating system including a kernel a set of associated local system files compatible with the processor, a method of providing at least some of the servers in the system with secure, executable, applications related to a service, wherein the applications are executed in a secure environment, wherein the applications each include an object executable by at least some of the different operating systems for performing a task related to the service, the method comprising:	To the extent the preamble is limiting, Microsoft and/or its customer practices, through the Accused Instrumentalities, in a system having a plurality of servers with operating systems that differ, operating in disparate computing environments, wherein each server includes a processor and an operating system including a kernel a set of associated local system files compatible with the processor, a method of providing at least some of the servers in the system with secure, executable, applications related to a service, wherein the applications are executed in a secure environment, wherein the applications each include an object executable by at least some of the different operating systems for performing a task related to the service, as claimed. For example, Azure Kubernetes Service runs on individual servers, each of which runs an independent operating system running either on bare metal, through an on-premises virtualized infrastructure, through one or more cloud services, or through any other supported deployment. In an exemplary deployment, two or more servers use different operating systems. The servers operate in disparate computing environments, including because each server is a stand-alone computer and/or each server is unrelated to the other servers due to having independent hardware and, in some instances, independent software. Microsoft requires and/or provides that each server includes a processor with one or more cores available to the OS kernel. Microsoft further requires and/or provides that each server has a supported operating system, which includes a kernel and associated local system files, configuration files, etc. In the infringing system, at least two servers have different operating systems. In at least some instances, Microsoft directly owns, operates, controls, and/or benefits from the claimed system and/or method. In other instances, Microsoft's customer makes and uses the system and/or method either by following Microsoft's direction and control, including Microsoft's documentation, or automatically through the ordinary and expected operation of Microsoft's software, or a combination thereof.

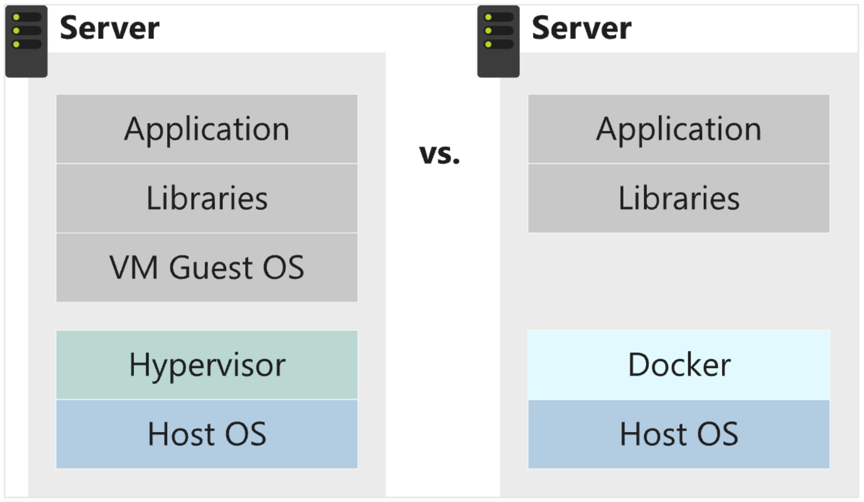
Claim 1	Accused Instrumentalities
	See claim limitations below. *See also, e.g.*: Azure Kubernetes Service (AKS) is a managed Kubernetes service that you can use to deploy and manage containerized applications. You need minimal container orchestration expertise to use AKS. AKS reduces the complexity and operational overhead of managing Kubernetes by offloading much of that responsibility to Azure. AKS is an ideal platform for deploying and managing containerized applications that require high availability, scalability, and portability, and for deploying applications to multiple regions, using open-source tools, and integrating with existing DevOps tools. https://learn.microsoft.com/en-us/azure/aks/what-is-aks **When to use AKS** The following list describes some common use cases for AKS: • Lift and shift to containers with AKS: Migrate existing applications to containers and run them in a fully managed Kubernetes environment. • Microservices with AKS: Simplify the deployment and management of microservices-based applications with streamlined horizontal scaling, self-healing, load balancing, and secret management. • Secure DevOps for AKS: Efficiently balance speed and security by implementing secure DevOps with Kubernetes. • Bursting from AKS with ACI: Use virtual nodes to provision pods inside ACI that start in seconds and scale to meet demand. • Machine learning model training with AKS: Train models using large datasets with familiar tools, such as TensorFlow and Kubeflow. • Data streaming with AKS: Ingest and process real-time data streams with millions of data points collected via sensors, and perform fast analyses and computations to develop insights into complex scenarios. • Using Windows containers on AKS: Run Windows Server containers on AKS to modernize your Windows applications and infrastructure. https://learn.microsoft.com/en-us/azure/aks/what-is-aks

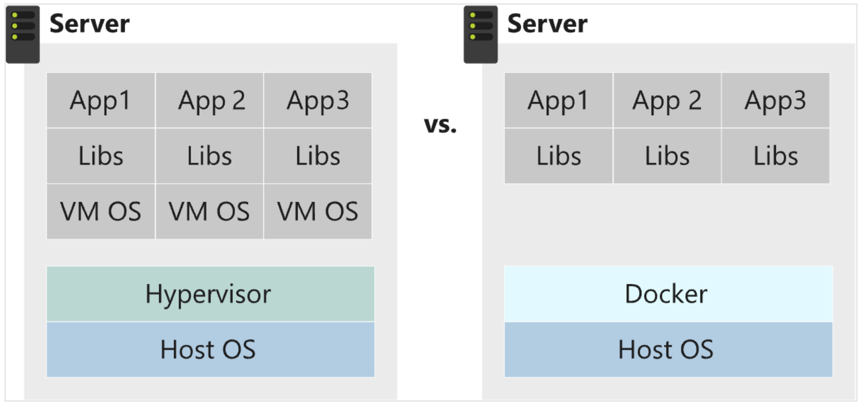
Claim 1	Accused Instrumentalities
	 Azure Arc-enabled Kubernetes allows you to attach Kubernetes clusters running anywhere so that you can manage and configure them in Azure. By managing all of your Kubernetes resources in a single control plane, you can enable a more consistent development and operation experience, helping you run cloud-native apps anywhere and on any Kubernetes platform. When the Azure Arc agents are deployed to the cluster, an outbound connection to Azure is initiated, using industry-standard SSL to secure data in transit. Clusters that you connect to Azure are represented as their own resources in Azure Resource Manager, and they can be organized using resource groups and tagging. https://learn.microsoft.com/en-us/azure/azure-arc/kubernetes/overview  Containers are becoming the preferred way to package, deploy, and manage cloud applications. Azure Container Instances offers the fastest and simplest way to run Linux or Windows containers in Azure, without having to manage any virtual machines and without having to adopt a higher-level service. ACI supports regular, confidential, and Spot containers. ACI can be used as single-instance or multi-instance via NGroups, or you can get orchestration capabilities by deploying pods in your Azure Kubernetes Service (AKS) cluster via virtual nodes on ACI. For even faster startup times, ACI supports standby pools. https://learn.microsoft.com/en-us/azure/container-instances/container-instances-overview

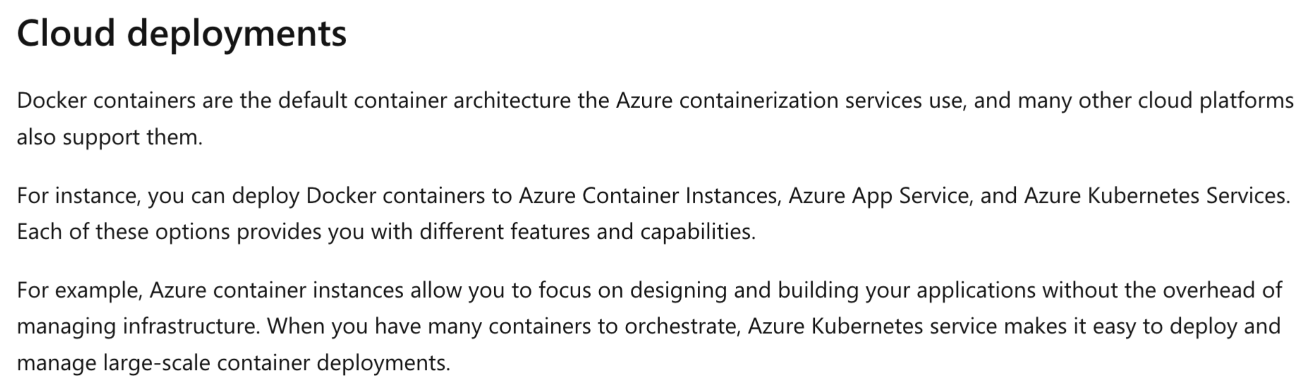
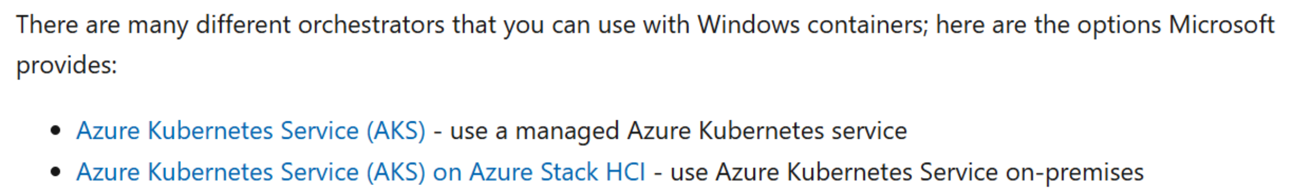
Claim 1	Accused Instrumentalities
	 Azure Container Apps is a serverless platform that allows you to maintain less infrastructure and save costs while running containerized applications. Instead of worrying about server configuration, container orchestration, and deployment details, Container Apps provides all the up-to-date server resources required to keep your applications stable and secure. Common uses of Azure Container Apps include: • Deploying API endpoints • Hosting background processing jobs • Handling event-driven processing • Running microservices https://learn.microsoft.com/en-us/azure/container-apps/overview

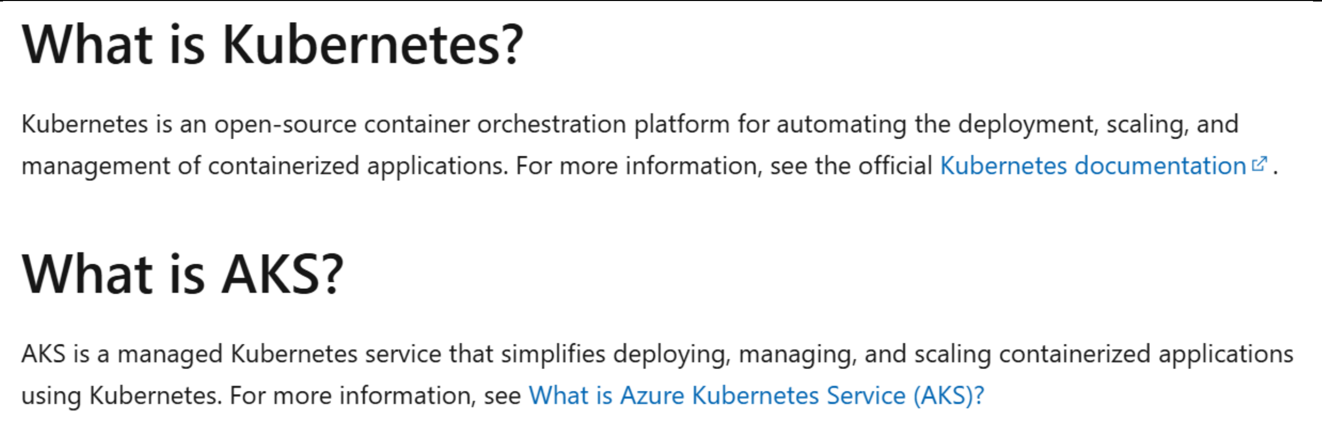
Claim 1	Accused Instrumentalities
	Applies to: Windows Server 2022, Windows Server 2019, Windows Server 2016 Containers are a technology for packaging and running Windows and Linux applications across diverse environments on-premises and in the cloud. Containers provide a lightweight, isolated environment that makes apps easier to develop, deploy, and manage. Containers start and stop quickly, making them ideal for apps that need to rapidly adapt to changing demand. The lightweight nature of containers also make them a useful tool for increasing the density and utilization of your infrastructure.  Anywhere On-premises Cloud Any app Monolith Microservice Any language Java .Net Python Node https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

Claim 1	Accused Instrumentalities
	 ## How containers work A container is an isolated, lightweight package for running an application on the host operating system. Containers build on top of the host operating system's kernel (which can be thought of as the buried plumbing of the operating system), as shown in the diagram below.  Container Container Apps Services Apps Services Apps Services OS Kernel Hardware While a container shares the host operating system's kernel, the container doesn't get unfettered access to it. Instead, the container gets an isolated–and in some cases virtualized–view of the system. For example, a container can access a virtualized version of the file system and registry, but any changes affect only the container and are discarded when it stops. To save data, the container can mount persistent storage such as an Azure Disk or a file share (including Azure Files). A container builds on top of the kernel, but the kernel doesn't provide all of the APIs and services an app needs to run–most of these are provided by system files (libraries) that run above the kernel in user mode. Because a container is isolated from the host's user mode environment, the container needs its own copy of these user mode system files, which are packaged into something known as a base image. The base image serves as the foundational layer upon which your container is built, providing it with operating system services not provided by the kernel. But we'll talk more about container images later. https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

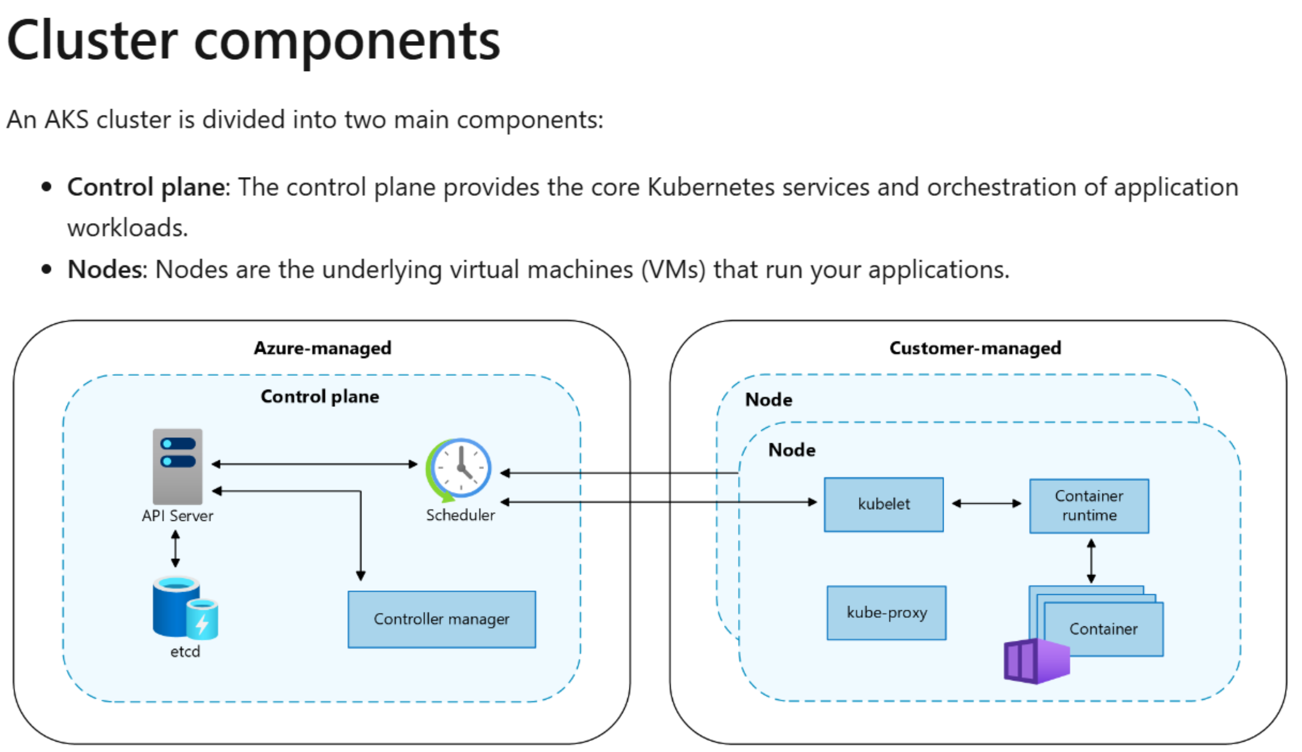
Claim 1	Accused Instrumentalities
	Docker benefits When we use Docker, we immediately get access to the benefits containerization offer. **Efficient hardware use** Containers run without using a virtual machine (VM). As we learned, the container relies on the host kernel for functions such as file system, network management, process scheduling, and memory management.  Server Application Libraries VM Guest OS Hypervisor Host OS vs. Server Application Libraries Docker Host OS Compared to a VM, we can see that a VM requires an OS installed to provide kernel functions to the running applications inside the VM. Keep in mind that the VM OS also requires disk space, memory, and CPU time. By removing the VM and the additional OS requirement, we can free resources on the host and use it for running other containers. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

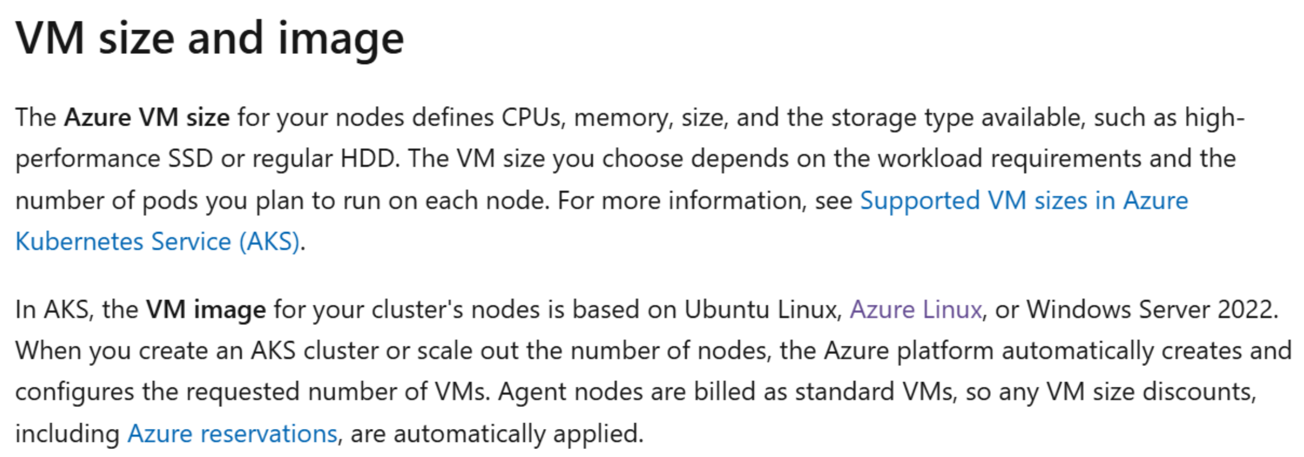
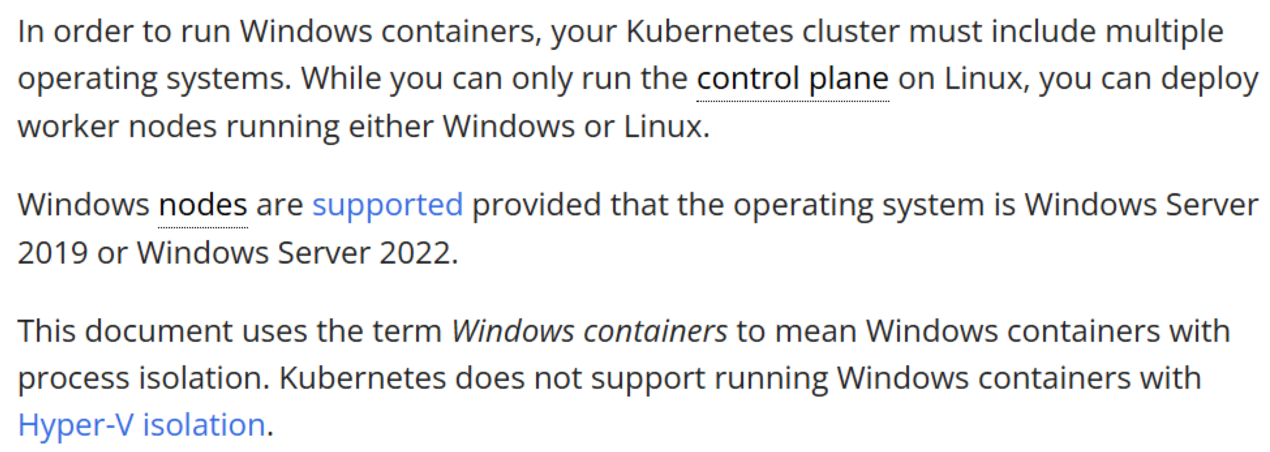
Claim 1	Accused Instrumentalities
	Container isolation Docker containers provide security features to run multiple containers simultaneously on the same host without affecting each other. As we learned, we can configure both data storage and network configuration to isolate our containers or share data and connectivity between specific containers. Let's compare this feature to using VMs.  Server App1 App 2 App3 Libs Libs Libs VM OS VM OS VM OS Hypervisor Host OS vs. Server App1 App 2 App3 Libs Libs Libs Docker Host OS Assume we have a physical host running two VMs. We have three applications that we want to run isolated from each other. We decide to deploy the first app onto VM1 and the second onto VM2 to separate the two apps from each other. If we now choose to install the third application, we'll need to install another VM to continue this pattern. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers **Application portability** Containers run almost everywhere: desktops, physical servers, VMs, and in the cloud. This runtime compatibility makes it easy to move containerized applications among different environments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

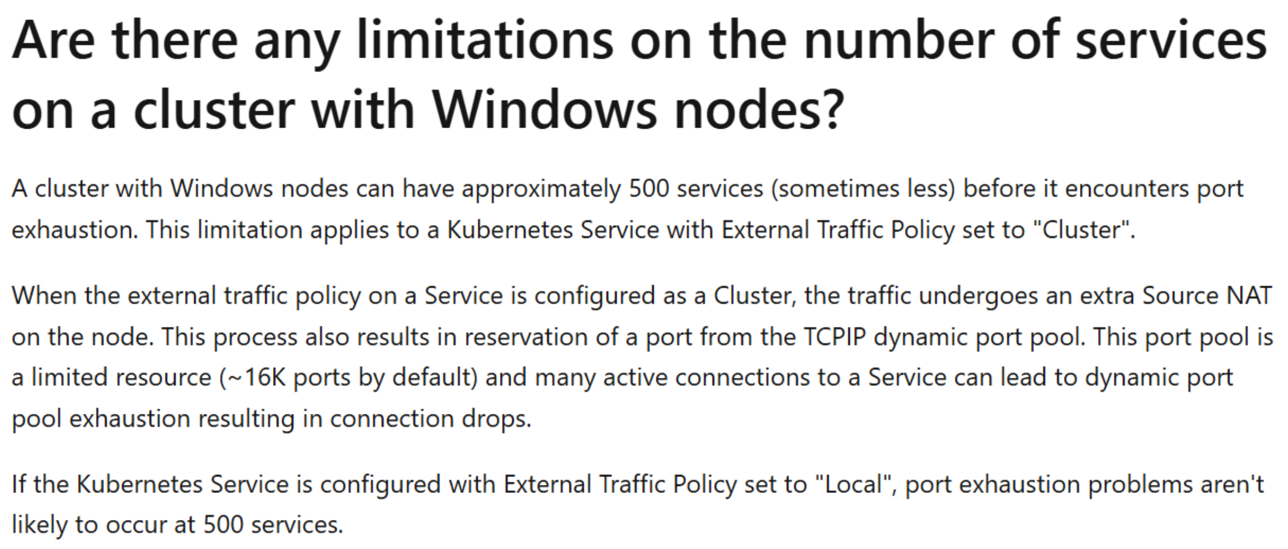
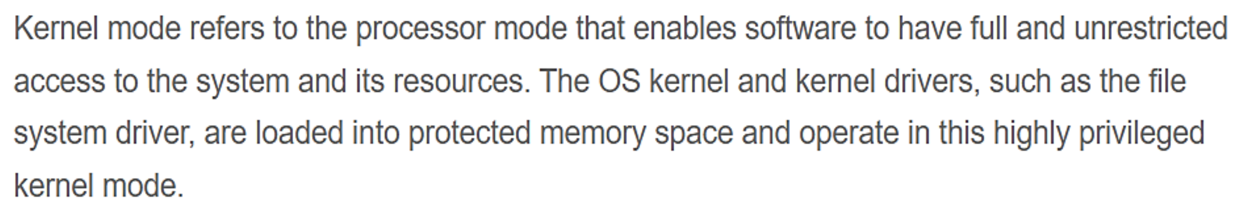
Claim 1	Accused Instrumentalities
	 ## Cloud deployments Docker containers are the default container architecture the Azure containerization services use, and many other cloud platforms also support them. For instance, you can deploy Docker containers to Azure Container Instances, Azure App Service, and Azure Kubernetes Services. Each of these options provides you with different features and capabilities. For example, Azure container instances allow you to focus on designing and building your applications without the overhead of managing infrastructure. When you have many containers to orchestrate, Azure Kubernetes service makes it easy to deploy and manage large-scale container deployments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers  There are many different orchestrators that you can use with Windows containers; here are the options Microsoft provides: • Azure Kubernetes Service (AKS) - use a managed Azure Kubernetes service • Azure Kubernetes Service (AKS) on Azure Stack HCI - use Azure Kubernetes Service on-premises https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

Claim 1	Accused Instrumentalities
	 **What is Kubernetes?** Kubernetes is an open-source container orchestration platform for automating the deployment, scaling, and management of containerized applications. For more information, see the official Kubernetes documentation. **What is AKS?** AKS is a managed Kubernetes service that simplifies deploying, managing, and scaling containerized applications using Kubernetes. For more information, see What is Azure Kubernetes Service (AKS)? https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts

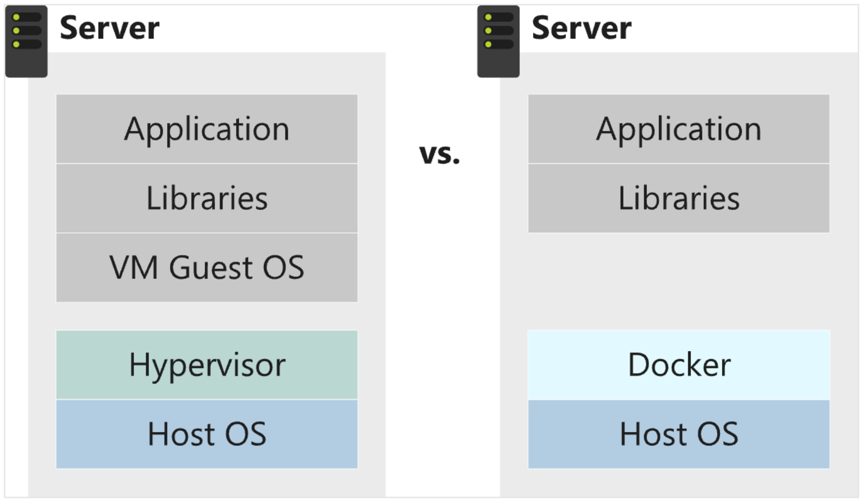
Case 4:24-cv-07003-JST Document 63-2 Filed 12/10/25 Page 13 of 73

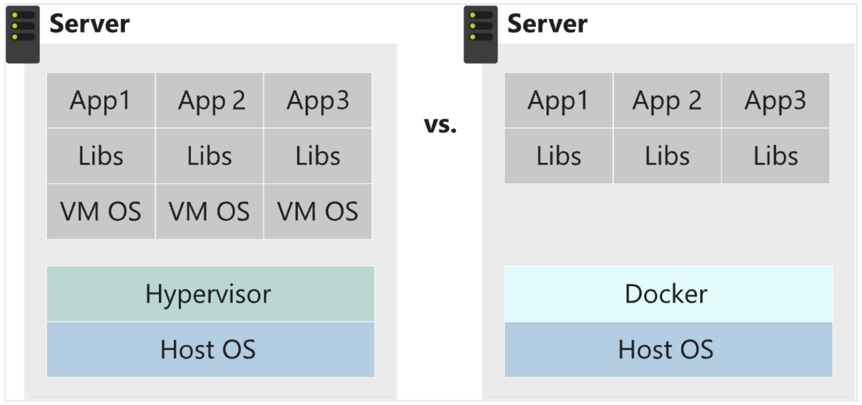
Claim 1	Accused Instrumentalities
	 ## Cluster components An AKS cluster is divided into two main components: • **Control plane**: The control plane provides the core Kubernetes services and orchestration of application workloads. • **Nodes**: Nodes are the underlying virtual machines (VMs) that run your applications. Azure-managed Control plane API Server Scheduler etcd Controller manager Customer-managed Node Node kubelet Container runtime kube-proxy Container https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts

Claim 1	Accused Instrumentalities
	 ## VM size and image The **Azure VM size** for your nodes defines CPUs, memory, size, and the storage type available, such as high-performance SSD or regular HDD. The VM size you choose depends on the workload requirements and the number of pods you plan to run on each node. For more information, see Supported VM sizes in Azure Kubernetes Service (AKS). In AKS, the **VM image** for your cluster's nodes is based on Ubuntu Linux, Azure Linux, or Windows Server 2022. When you create an AKS cluster or scale out the number of nodes, the Azure platform automatically creates and configures the requested number of VMs. Agent nodes are billed as standard VMs, so any VM size discounts, including Azure reservations, are automatically applied. https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts  In order to run Windows containers, your Kubernetes cluster must include multiple operating systems. While you can only run the control plane on Linux, you can deploy worker nodes running either Windows or Linux. Windows nodes are supported provided that the operating system is Windows Server 2019 or Windows Server 2022. This document uses the term *Windows containers* to mean Windows containers with process isolation. Kubernetes does not support running Windows containers with Hyper-V isolation. https://kubernetes.io/docs/concepts/windows/intro/

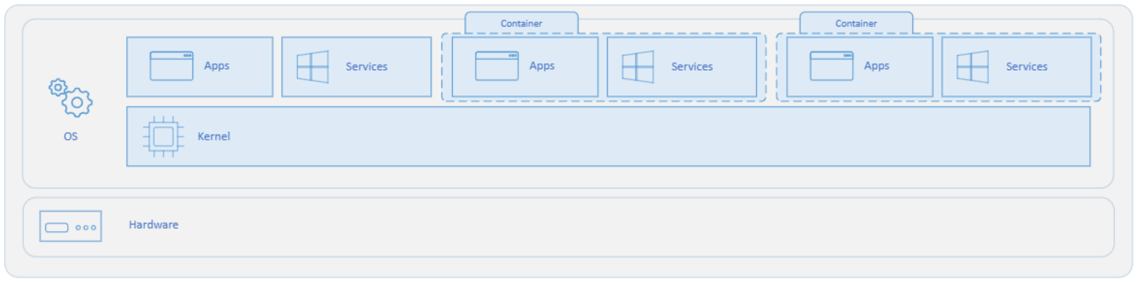
Claim 1	Accused Instrumentalities
	 **Are there any limitations on the number of services on a cluster with Windows nodes?** A cluster with Windows nodes can have approximately 500 services (sometimes less) before it encounters port exhaustion. This limitation applies to a Kubernetes Service with External Traffic Policy set to "Cluster". When the external traffic policy on a Service is configured as a Cluster, the traffic undergoes an extra Source NAT on the node. This process also results in reservation of a port from the TCPIP dynamic port pool. This port pool is a limited resource (~16K ports by default) and many active connections to a Service can lead to dynamic port pool exhaustion resulting in connection drops. If the Kubernetes Service is configured with External Traffic Policy set to "Local", port exhaustion problems aren't likely to occur at 500 services. https://learn.microsoft.com/en-us/azure/aks/windows-faq  Kernel mode refers to the processor mode that enables software to have full and unrestricted access to the system and its resources. The OS kernel and kernel drivers, such as the file system driver, are loaded into protected memory space and operate in this highly privileged kernel mode. https://www.techtarget.com/searchdatacenter/definition/kernel

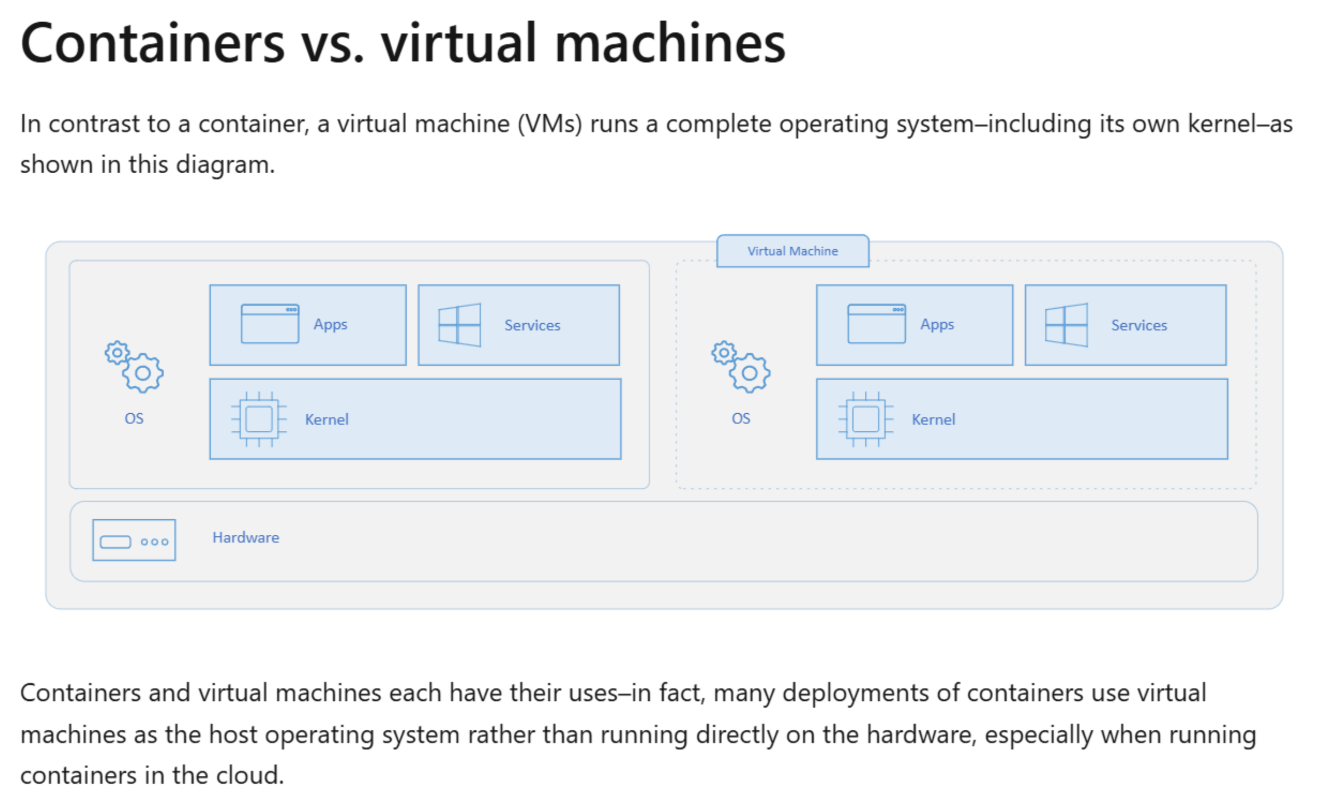
Claim 1	Accused Instrumentalities
[1a] storing in memory accessible to at least some of the servers a plurality of secure containers of application software, each container comprising one or more of the executable applications and a set of associated system files required to execute the one or more applications, for use with a local kernel residing permanently on one of the servers;	The method practiced by Microsoft and/or its customer through the Accused Instrumentalities includes a step of storing in memory accessible to at least some of the servers a plurality of secure containers of application software, each container comprising one or more of the executable applications and a set of associated system files required to execute the one or more applications, for use with a local kernel residing permanently on one of the servers. For example, OCI and/or OKE stores application containers, sometimes called Docker containers, container images, Kubernetes containers, or Kubernetes pods, in persistent storage available to each node running the application. The terms "node" and "host" are both used to refer to the claimed server. The container might be in a format defined by the Open Container Initiative. This storage may be physically attached to the server or connected through any supported interconnect, including over a network. In addition to the application software, each container includes associated system files, including a Linux user space required to execute the application, for example including runtime linked libraries (*e.g.* .dll/.so files), configuration files, Windows services, etc. necessary for the application. For example, the container includes a base OS image provided by Microsoft or by a third party. The container is for use with the host kernel, for example because the application(s) and container libraries are linked against the Linux kernel, and the supported host operating systems also use the Linux kernel, which has a stable binary interface. In another example, the container is for use with the host kernel because the application(s) and Windows services within the container are compatible with the host Windows operating system and kernel. The containers are secure containers as claimed. For example, the data within an individual container is insulated from the effects of other containers except to the extent the container is specifically configured to allow other containers to modify its data, for example using a shared volume. *See, e.g.*:

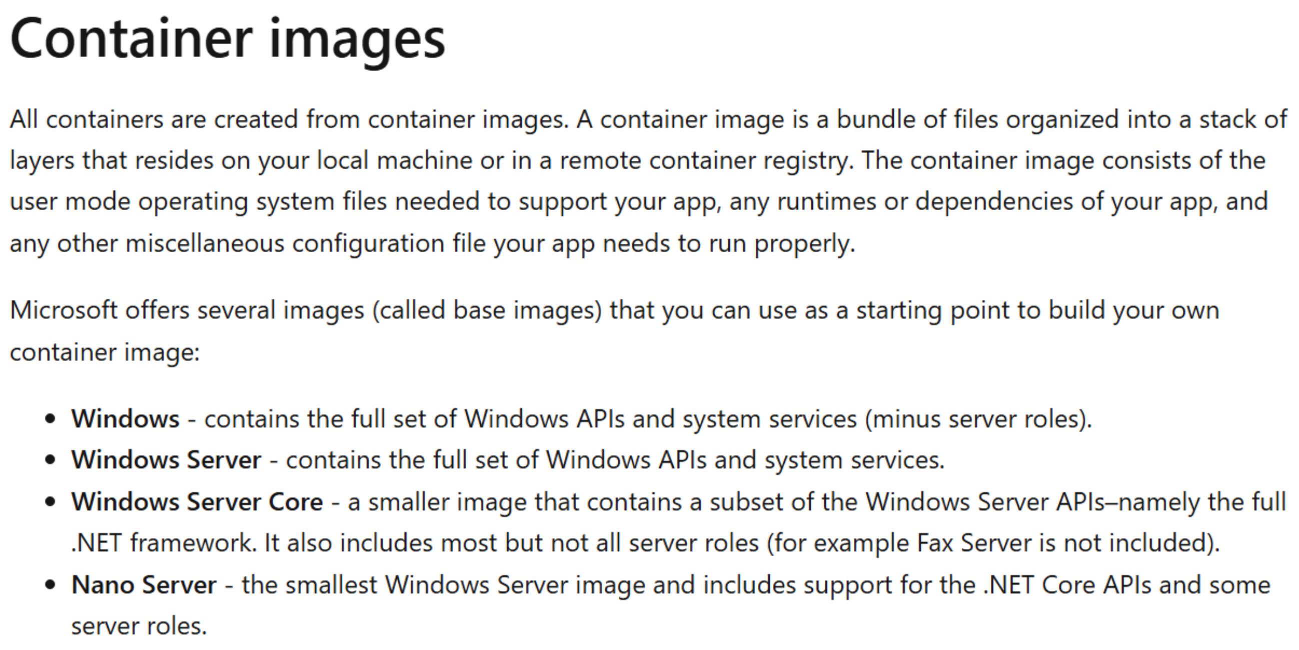
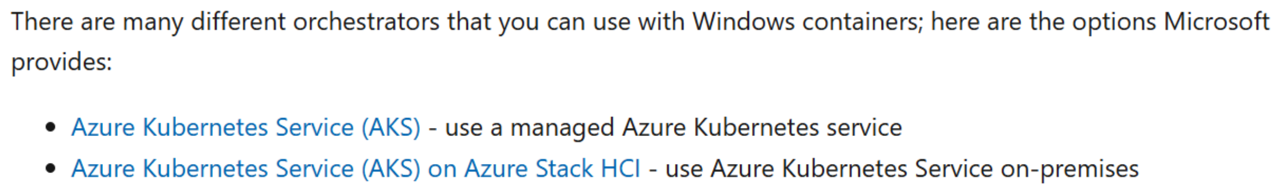
Claim 1	Accused Instrumentalities
	Docker benefits When we use Docker, we immediately get access to the benefits containerization offer. **Efficient hardware use** Containers run without using a virtual machine (VM). As we learned, the container relies on the host kernel for functions such as file system, network management, process scheduling, and memory management.  Server Application Libraries VM Guest OS Hypervisor Host OS vs. Server Application Libraries Docker Host OS Compared to a VM, we can see that a VM requires an OS installed to provide kernel functions to the running applications inside the VM. Keep in mind that the VM OS also requires disk space, memory, and CPU time. By removing the VM and the additional OS requirement, we can free resources on the host and use it for running other containers. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

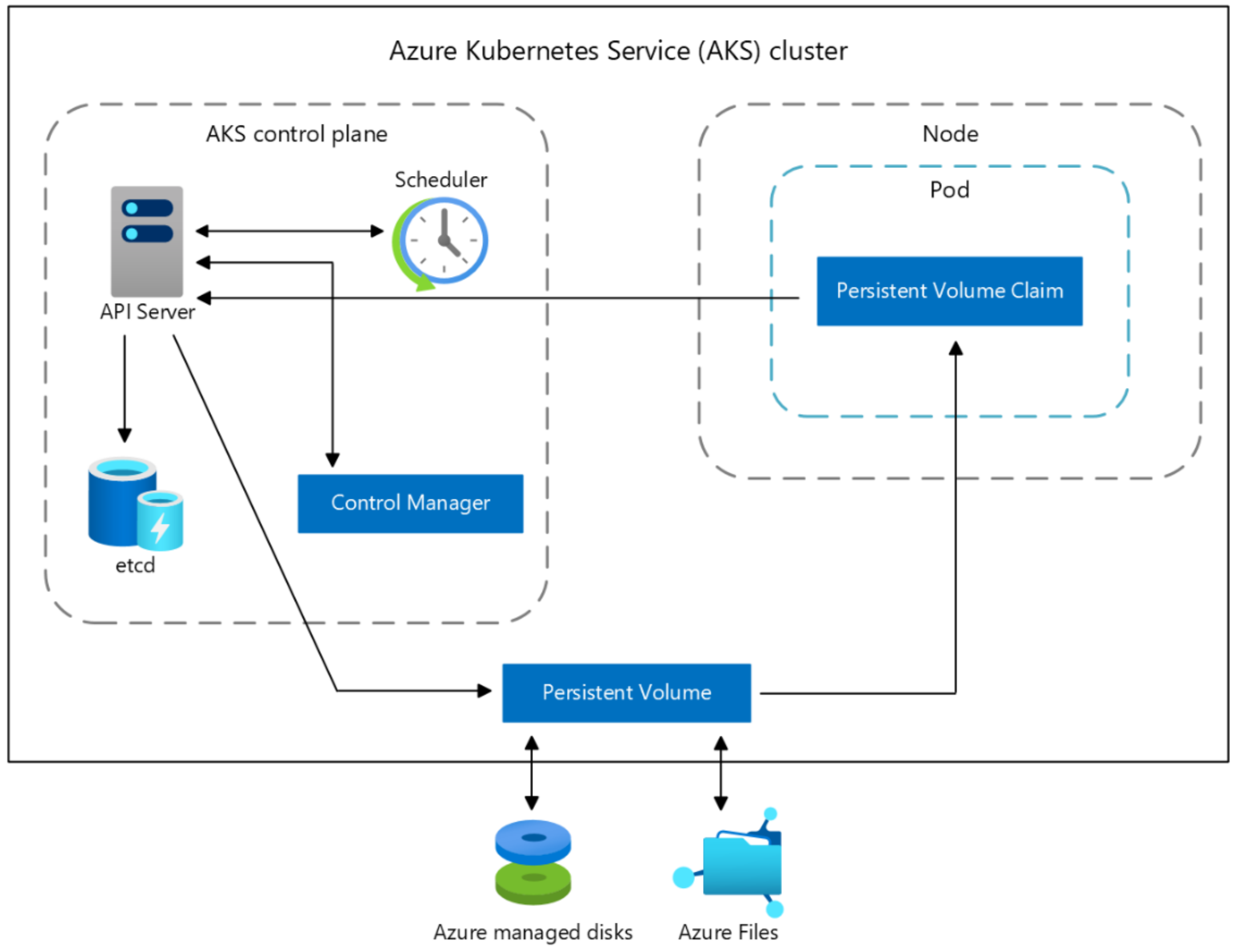
Claim 1	Accused Instrumentalities
	Container isolation Docker containers provide security features to run multiple containers simultaneously on the same host without affecting each other. As we learned, we can configure both data storage and network configuration to isolate our containers or share data and connectivity between specific containers. Let's compare this feature to using VMs.  Server App1 App 2 App3 Libs Libs Libs VM OS VM OS VM OS Hypervisor Host OS vs. Server App1 App 2 App3 Libs Libs Libs Docker Host OS Assume we have a physical host running two VMs. We have three applications that we want to run isolated from each other. We decide to deploy the first app onto VM1 and the second onto VM2 to separate the two apps from each other. If we now choose to install the third application, we'll need to install another VM to continue this pattern. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers **Application portability** Containers run almost everywhere: desktops, physical servers, VMs, and in the cloud. This runtime compatibility makes it easy to move containerized applications among different environments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

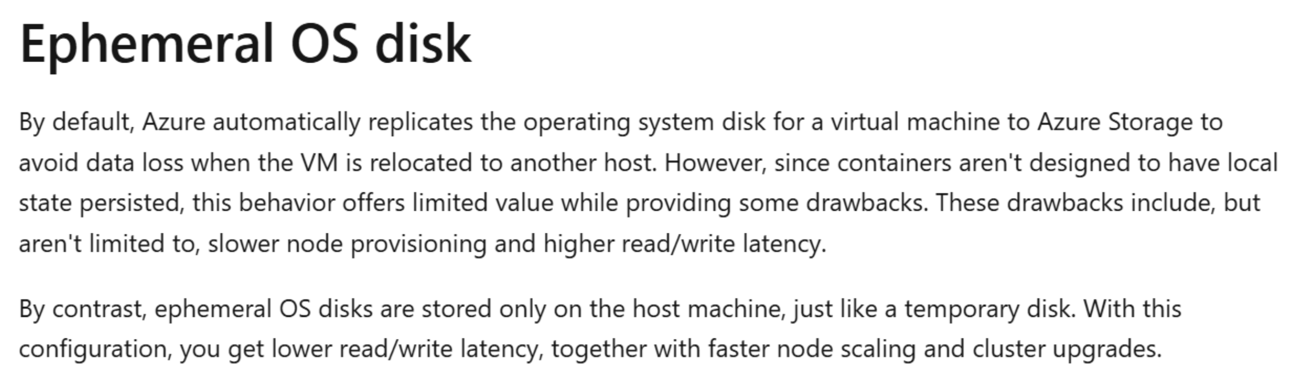
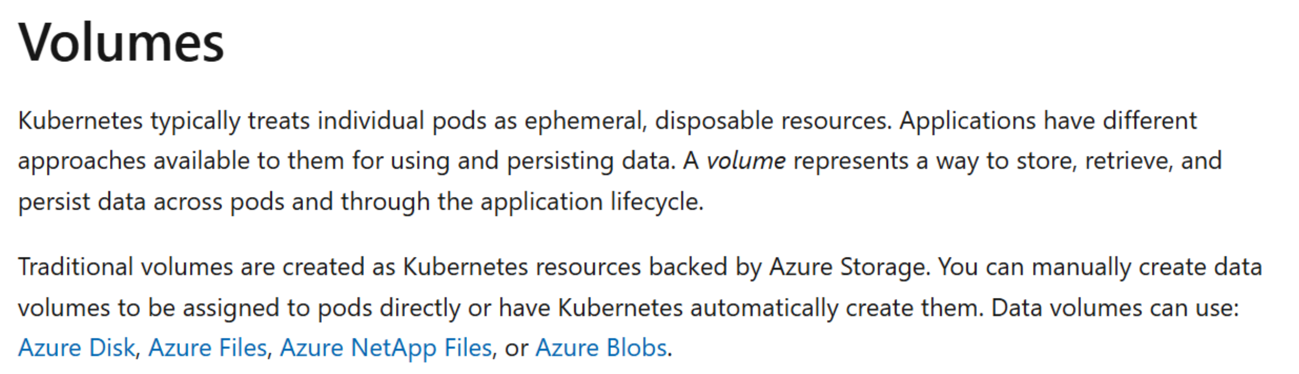
Claim 1	Accused Instrumentalities
	Cloud deployments Docker containers are the default container architecture the Azure containerization services use, and many other cloud platforms also support them. For instance, you can deploy Docker containers to Azure Container Instances, Azure App Service, and Azure Kubernetes Services. Each of these options provides you with different features and capabilities. For example, Azure container instances allow you to focus on designing and building your applications without the overhead of managing infrastructure. When you have many containers to orchestrate, Azure Kubernetes service makes it easy to deploy and manage large-scale container deployments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers • **Deploy containers at scale on Azure** or other clouds: ○ Pull your app (container image) from a container registry, such as the Azure Container Registry, and then deploy and manage it at scale using an orchestrator such as Azure Kubernetes Service (AKS). ○ Azure Kubernetes Service deploys containers to Azure virtual machines and manages them at scale, whether that's dozens of containers, hundreds, or even thousands. The Azure virtual machines run either a customized Windows Server image (if you're deploying a Windows-based app), or a customized Ubuntu Linux image (if you're deploying a Linux-based app). https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

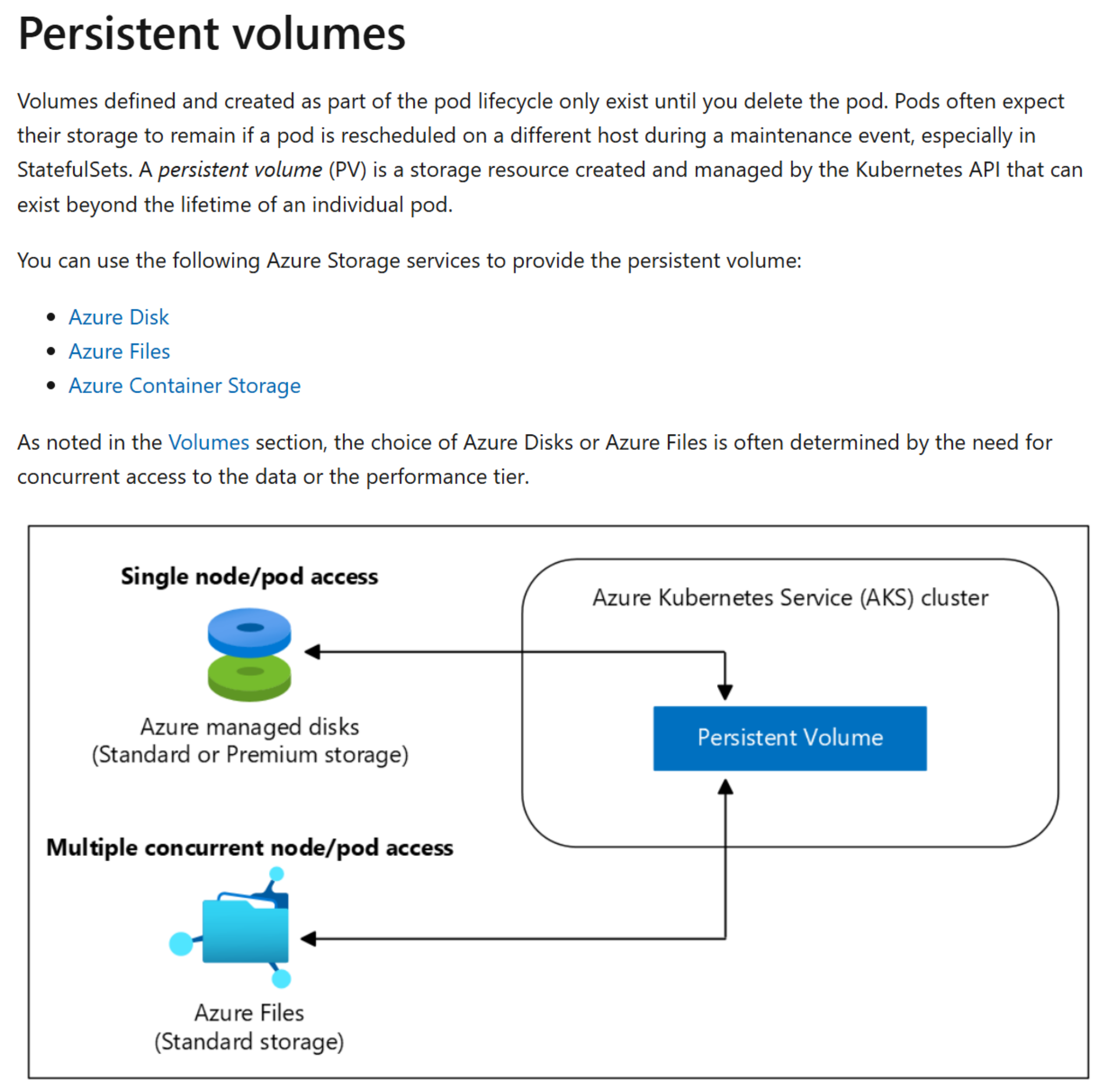
Claim 1	Accused Instrumentalities
	## How containers work A container is an isolated, lightweight package for running an application on the host operating system. Containers build on top of the host operating system's kernel (which can be thought of as the buried plumbing of the operating system), as shown in the diagram below.  Container Container Apps Services Apps Services Apps Services OS Kernel Hardware While a container shares the host operating system's kernel, the container doesn't get unfettered access to it. Instead, the container gets an isolated–and in some cases virtualized–view of the system. For example, a container can access a virtualized version of the file system and registry, but any changes affect only the container and are discarded when it stops. To save data, the container can mount persistent storage such as an Azure Disk or a file share (including Azure Files). A container builds on top of the kernel, but the kernel doesn't provide all of the APIs and services an app needs to run–most of these are provided by system files (libraries) that run above the kernel in user mode. Because a container is isolated from the host's user mode environment, the container needs its own copy of these user mode system files, which are packaged into something known as a base image. The base image serves as the foundational layer upon which your container is built, providing it with operating system services not provided by the kernel. But we'll talk more about container images later. https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

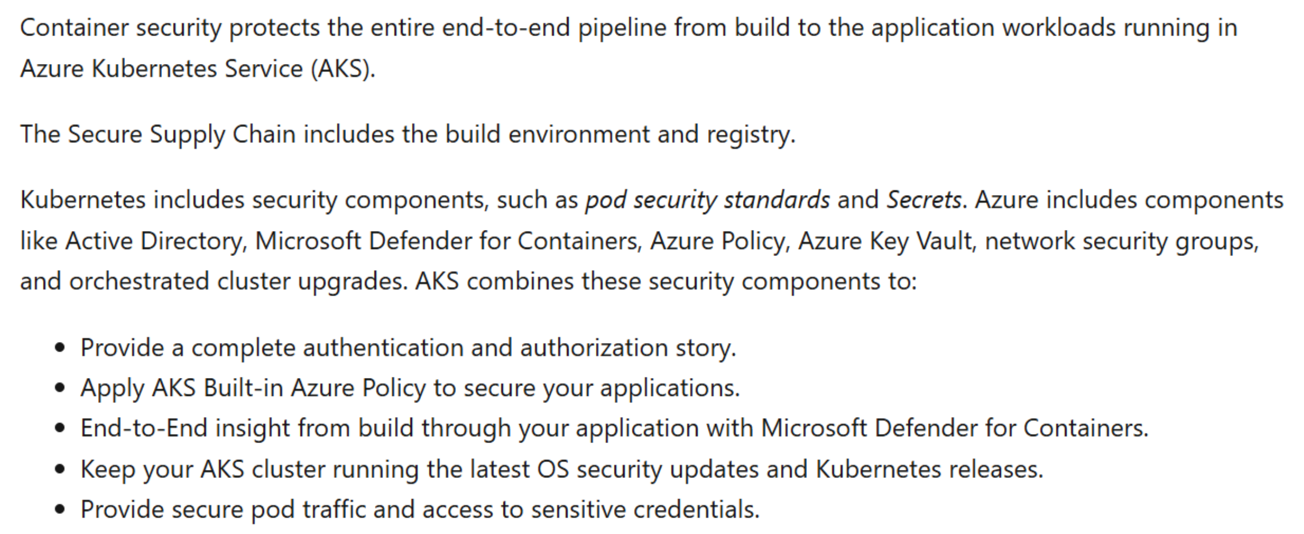
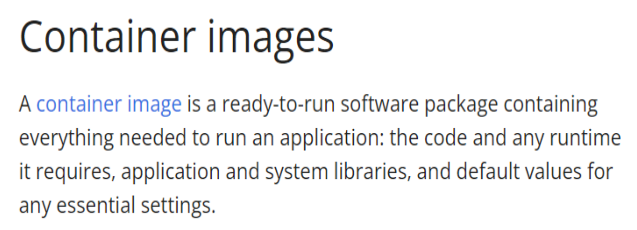
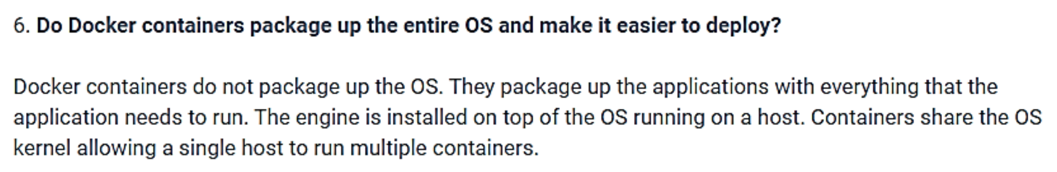
Claim 1	Accused Instrumentalities
	 ## Containers vs. virtual machines In contrast to a container, a virtual machine (VMs) runs a complete operating system–including its own kernel–as shown in this diagram. Virtual Machine Apps Services OS Kernel Apps Services OS Kernel Hardware Containers and virtual machines each have their uses–in fact, many deployments of containers use virtual machines as the host operating system rather than running directly on the hardware, especially when running containers in the cloud. https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

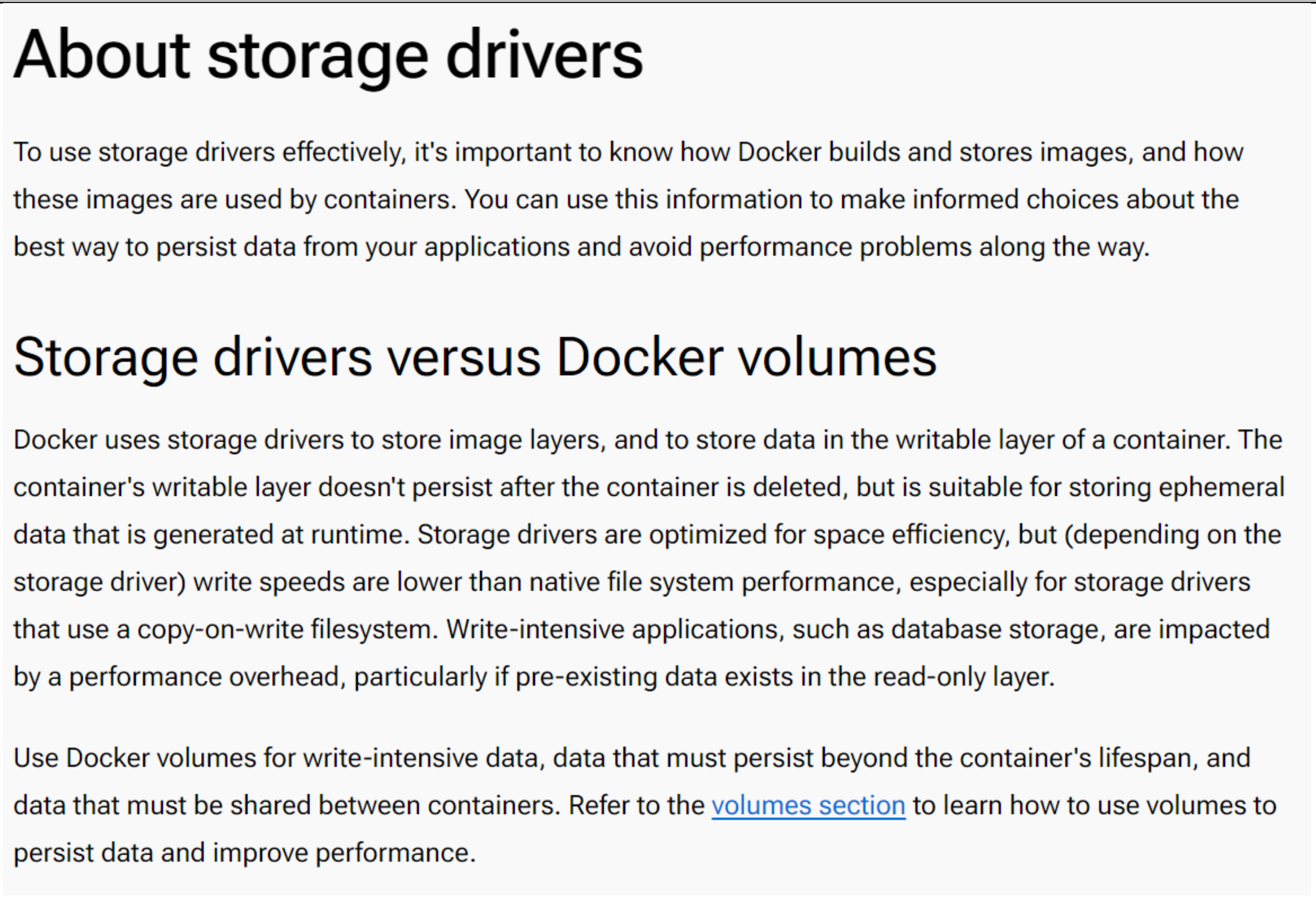
Claim 1	Accused Instrumentalities
	 ## Container images All containers are created from container images. A container image is a bundle of files organized into a stack of layers that resides on your local machine or in a remote container registry. The container image consists of the user mode operating system files needed to support your app, any runtimes or dependencies of your app, and any other miscellaneous configuration file your app needs to run properly. Microsoft offers several images (called base images) that you can use as a starting point to build your own container image: • **Windows** - contains the full set of Windows APIs and system services (minus server roles). • **Windows Server** - contains the full set of Windows APIs and system services. • **Windows Server Core** - a smaller image that contains a subset of the Windows Server APIs–namely the full .NET framework. It also includes most but not all server roles (for example Fax Server is not included). • **Nano Server** - the smallest Windows Server image and includes support for the .NET Core APIs and some server roles. https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/  There are many different orchestrators that you can use with Windows containers; here are the options Microsoft provides: • Azure Kubernetes Service (AKS) - use a managed Azure Kubernetes service • Azure Kubernetes Service (AKS) on Azure Stack HCI - use Azure Kubernetes Service on-premises https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

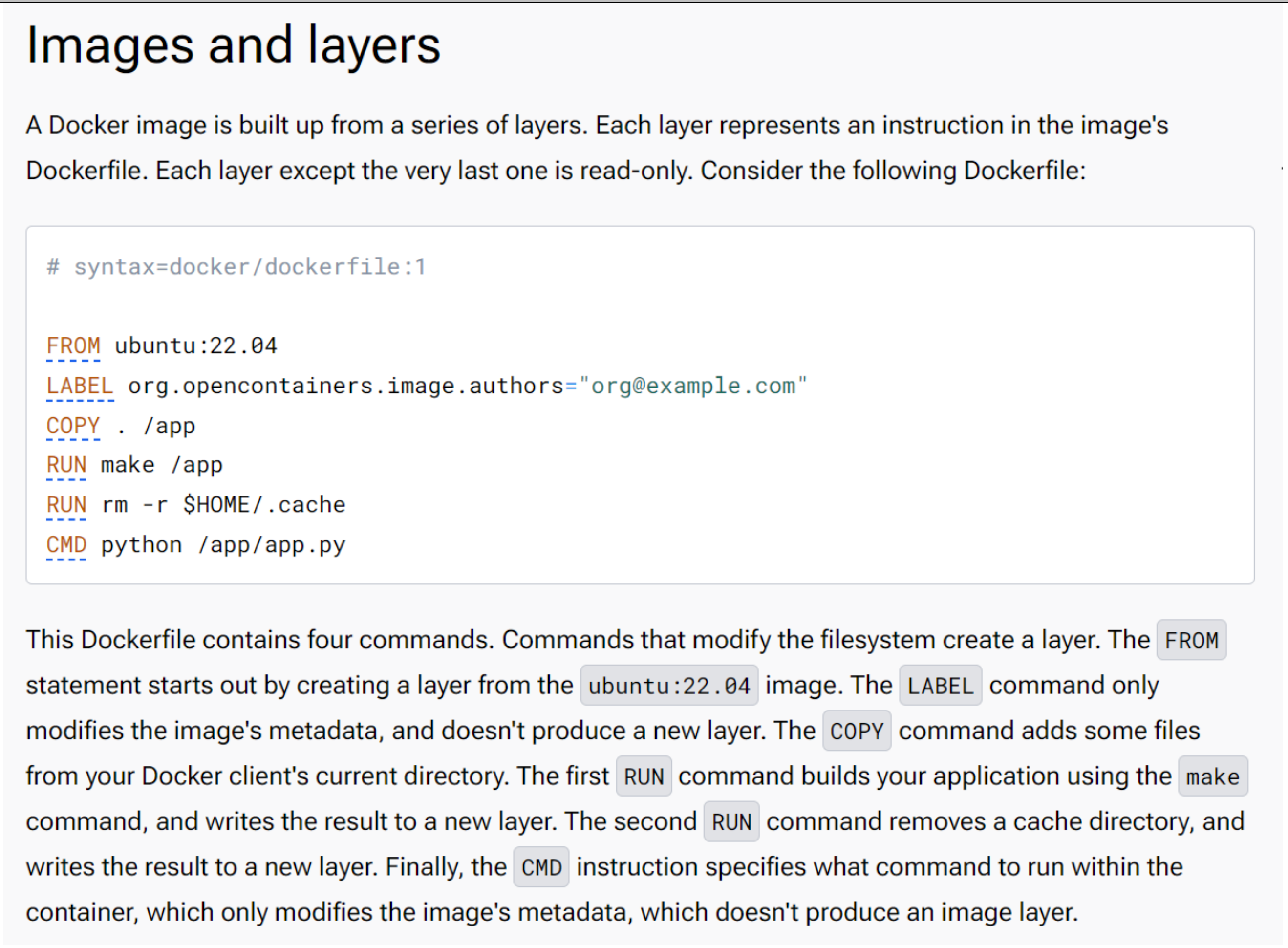
Claim 1	Accused Instrumentalities
	 Azure Kubernetes Service (AKS) cluster AKS control plane Scheduler API Server etcd Control Manager Node Pod Persistent Volume Claim Persistent Volume Azure managed disks Azure Files https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Claim 1	Accused Instrumentalities
	 ## Ephemeral OS disk By default, Azure automatically replicates the operating system disk for a virtual machine to Azure Storage to avoid data loss when the VM is relocated to another host. However, since containers aren't designed to have local state persisted, this behavior offers limited value while providing some drawbacks. These drawbacks include, but aren't limited to, slower node provisioning and higher read/write latency. By contrast, ephemeral OS disks are stored only on the host machine, just like a temporary disk. With this configuration, you get lower read/write latency, together with faster node scaling and cluster upgrades. https://learn.microsoft.com/en-us/azure/aks/concepts-storage  ## Volumes Kubernetes typically treats individual pods as ephemeral, disposable resources. Applications have different approaches available to them for using and persisting data. A *volume* represents a way to store, retrieve, and persist data across pods and through the application lifecycle. Traditional volumes are created as Kubernetes resources backed by Azure Storage. You can manually create data volumes to be assigned to pods directly or have Kubernetes automatically create them. Data volumes can use: Azure Disk, Azure Files, Azure NetApp Files, or Azure Blobs. https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Claim 1	Accused Instrumentalities
	 **Persistent volumes** Volumes defined and created as part of the pod lifecycle only exist until you delete the pod. Pods often expect their storage to remain if a pod is rescheduled on a different host during a maintenance event, especially in StatefulSets. A *persistent volume* (PV) is a storage resource created and managed by the Kubernetes API that can exist beyond the lifetime of an individual pod. You can use the following Azure Storage services to provide the persistent volume: • Azure Disk • Azure Files • Azure Container Storage As noted in the Volumes section, the choice of Azure Disks or Azure Files is often determined by the need for concurrent access to the data or the performance tier. Single node/pod access Azure managed disks (Standard or Premium storage) Azure Kubernetes Service (AKS) cluster Persistent Volume Multiple concurrent node/pod access Azure Files (Standard storage) https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Claim 1	Accused Instrumentalities
	 Container security protects the entire end-to-end pipeline from build to the application workloads running in Azure Kubernetes Service (AKS). The Secure Supply Chain includes the build environment and registry. Kubernetes includes security components, such as *pod security standards* and *Secrets*. Azure includes components like Active Directory, Microsoft Defender for Containers, Azure Policy, Azure Key Vault, network security groups, and orchestrated cluster upgrades. AKS combines these security components to: • Provide a complete authentication and authorization story. • Apply AKS Built-in Azure Policy to secure your applications. • End-to-End insight from build through your application with Microsoft Defender for Containers. • Keep your AKS cluster running the latest OS security updates and Kubernetes releases. • Provide secure pod traffic and access to sensitive credentials. https://learn.microsoft.com/en-us/azure/aks/concepts-security  Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/  6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/

Claim 1	Accused Instrumentalities
	 **About storage drivers** To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	
	https://docs.docker.com/storage/storagedriver/

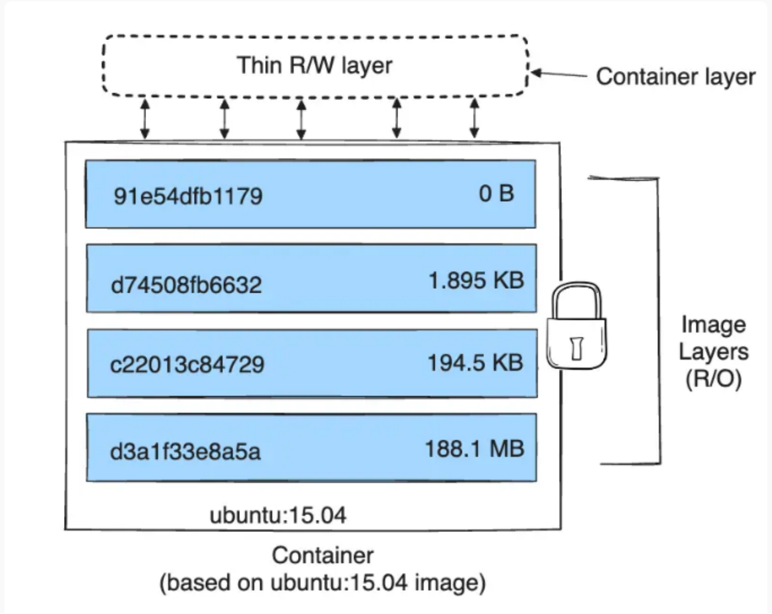
Images and layers

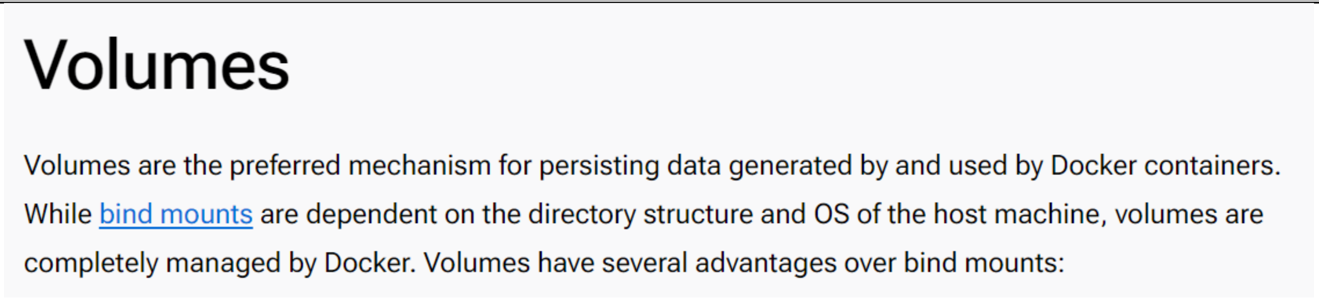
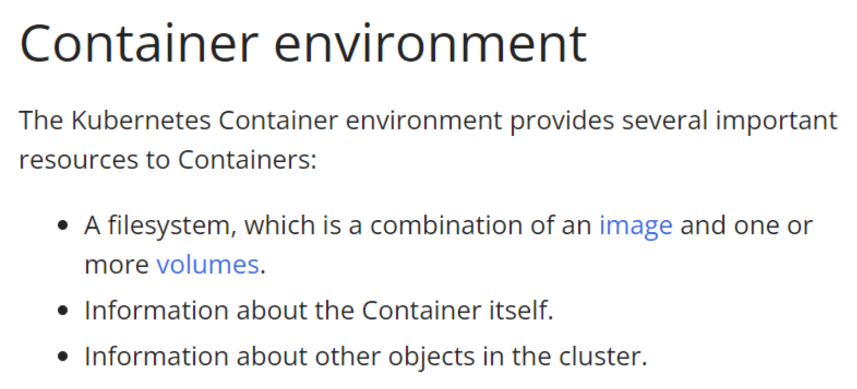
A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

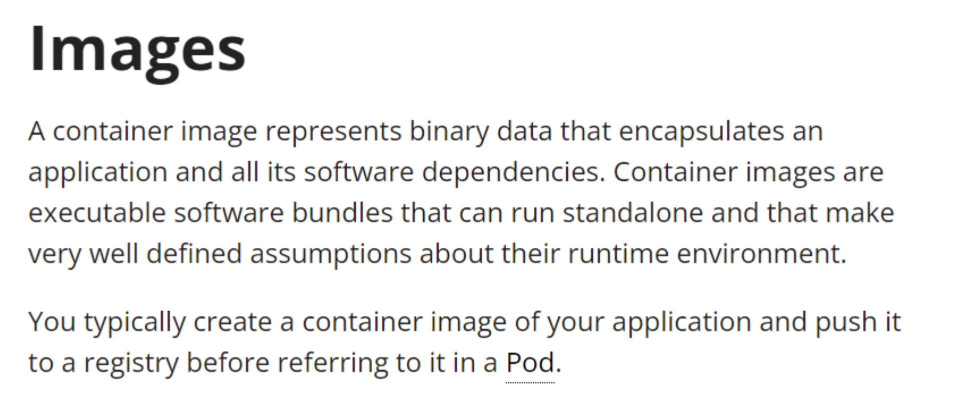
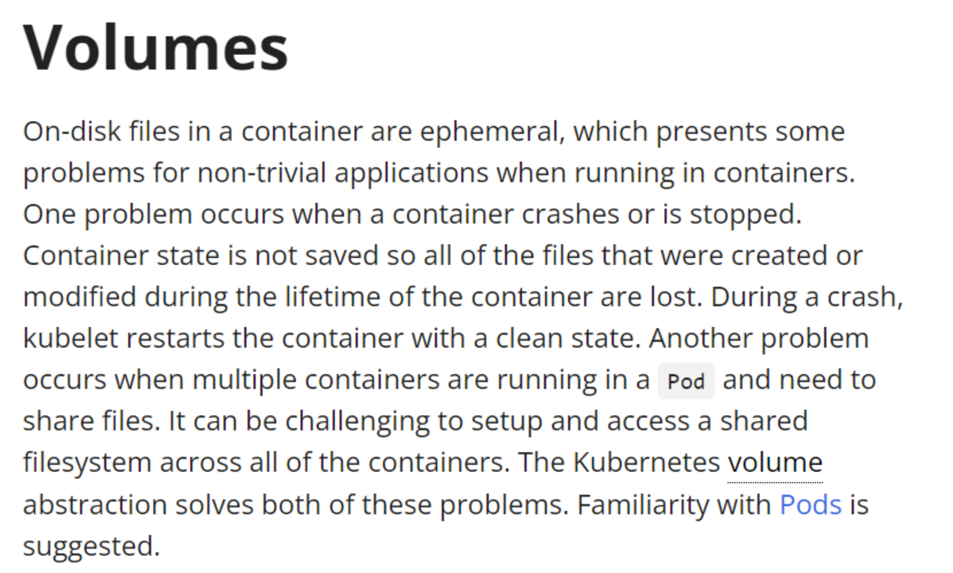
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

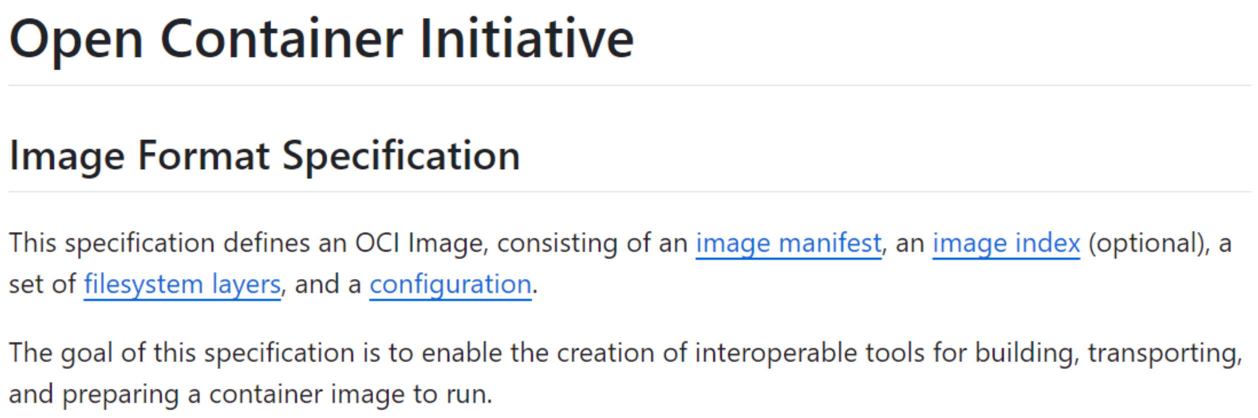
This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

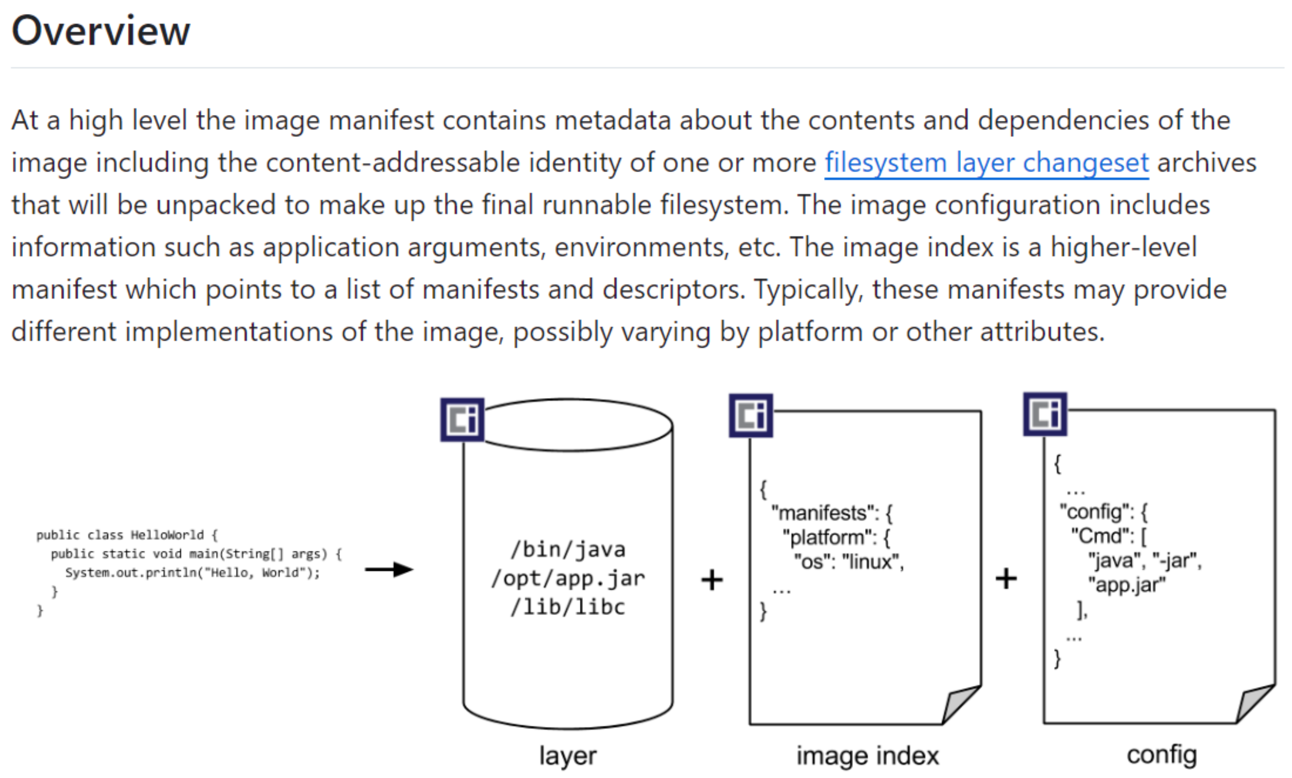
Case 5:24-cv-04740-PCP Document 63-2 Filed 01/21/25 Page 28 of 72

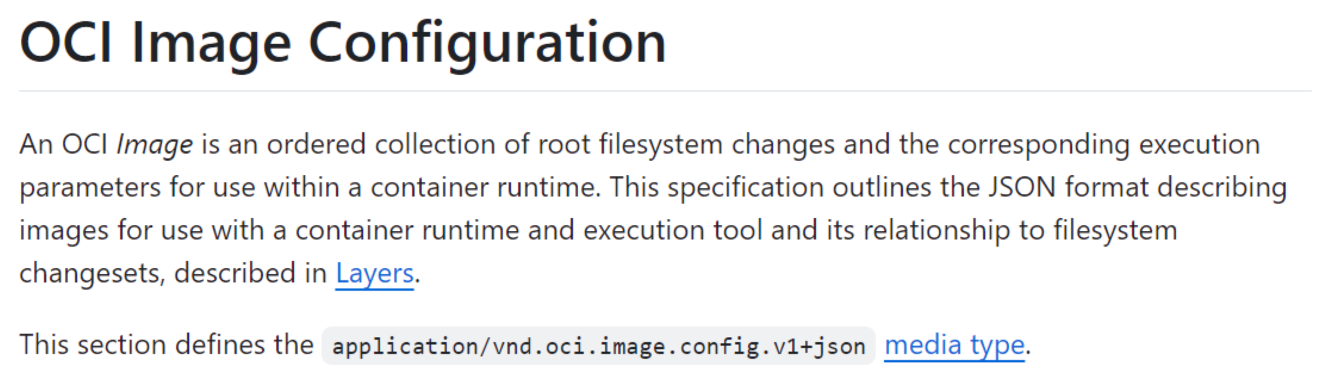
Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 Image Layers (R/O) Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

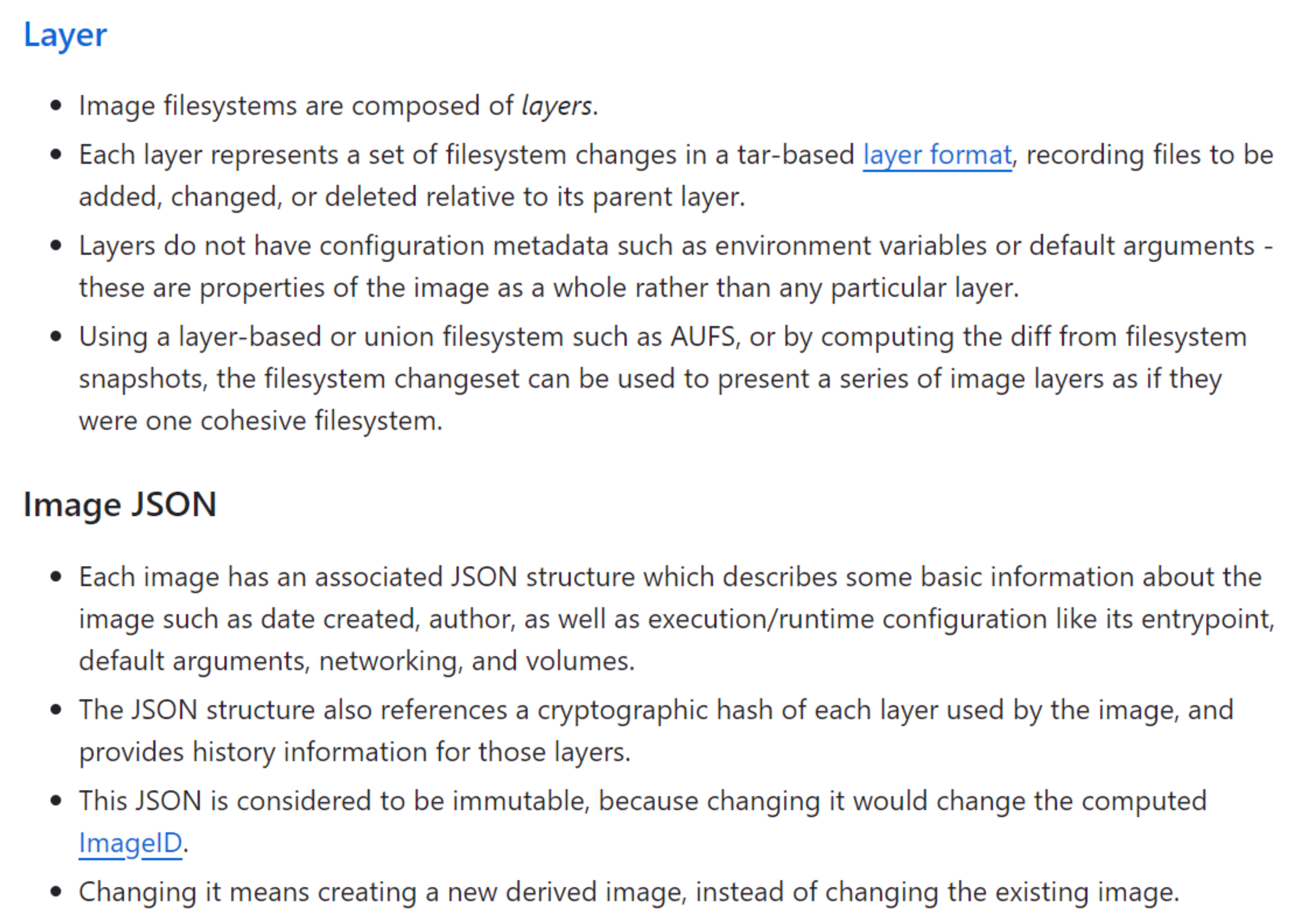
Claim 1	Accused Instrumentalities
	 ## Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/  ## Container environment The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

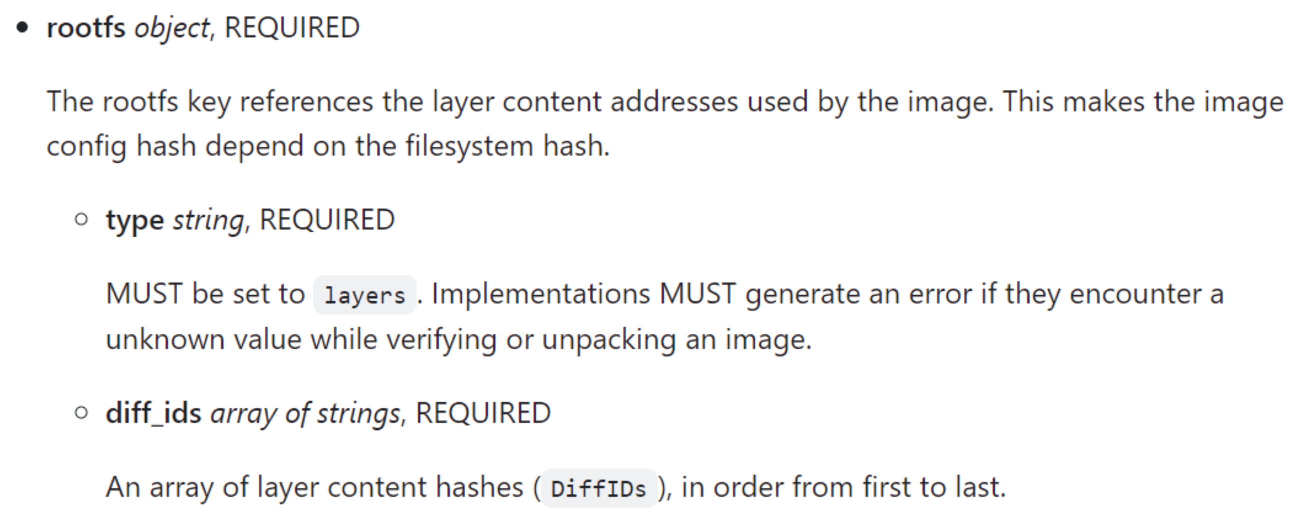
Claim 1	Accused Instrumentalities
	 **Images** A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/  **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

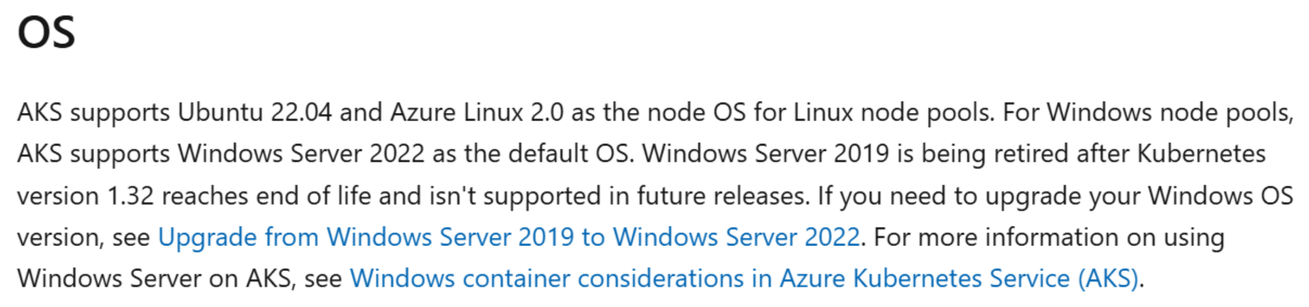
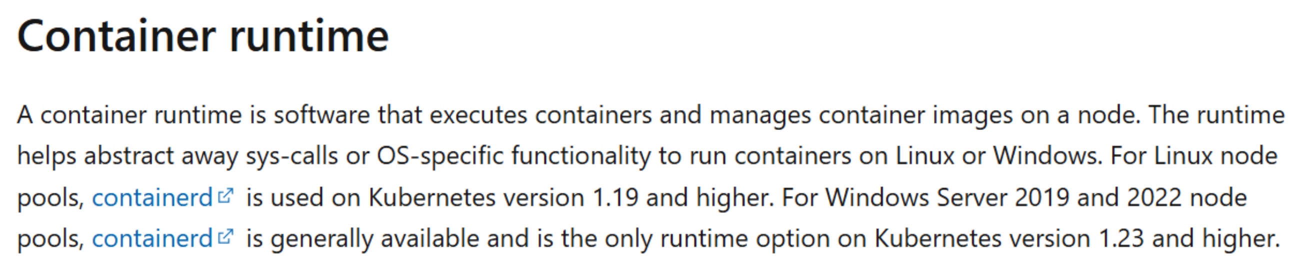
Claim 1	Accused Instrumentalities
	 Open Container Initiative Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

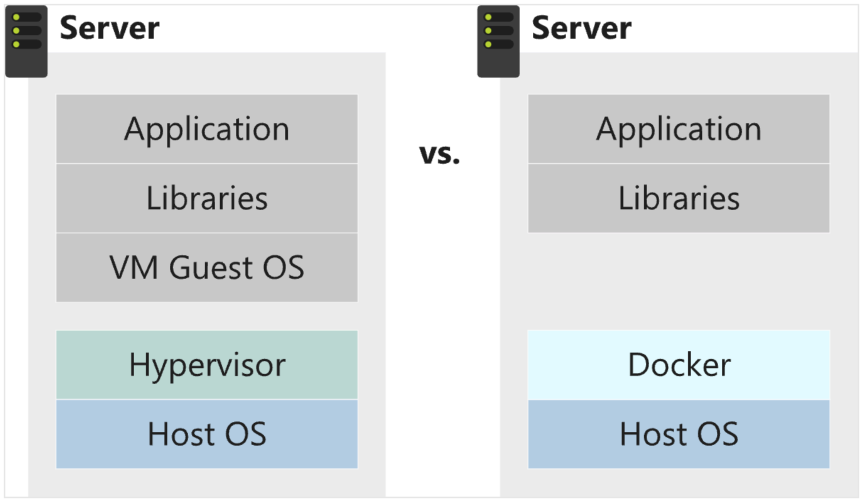
Claim 1	Accused Instrumentalities
	 ## Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes. public class HelloWorld { public static void main(String[] args) { System.out.println("Hello, World"); } } /bin/java /opt/app.jar /lib/libc + { "manifests": { "platform": { "os": "linux", ... } + { ... "config": { "Cmd": ["java", "-jar", "app.jar"], ... } layer image index config https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

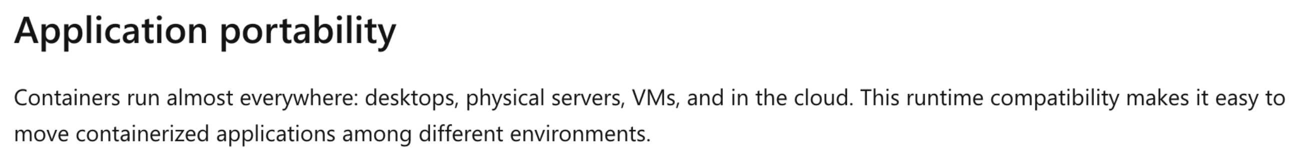
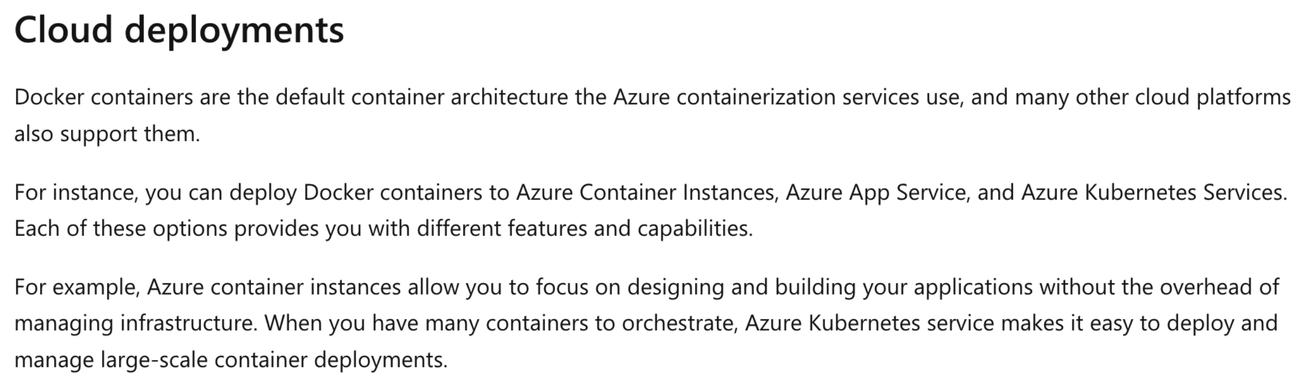
Claim 1	Accused Instrumentalities
	 OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

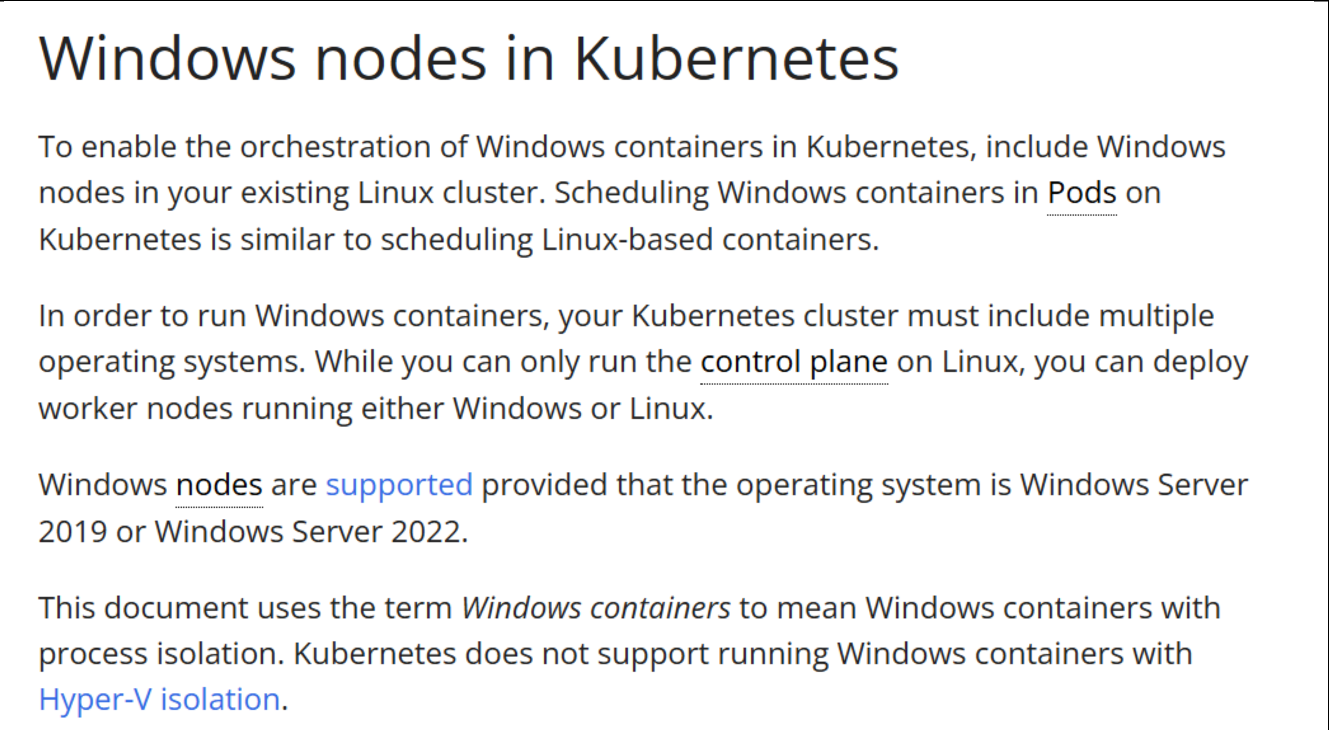
Claim 1	Accused Instrumentalities
	 **Layer** • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

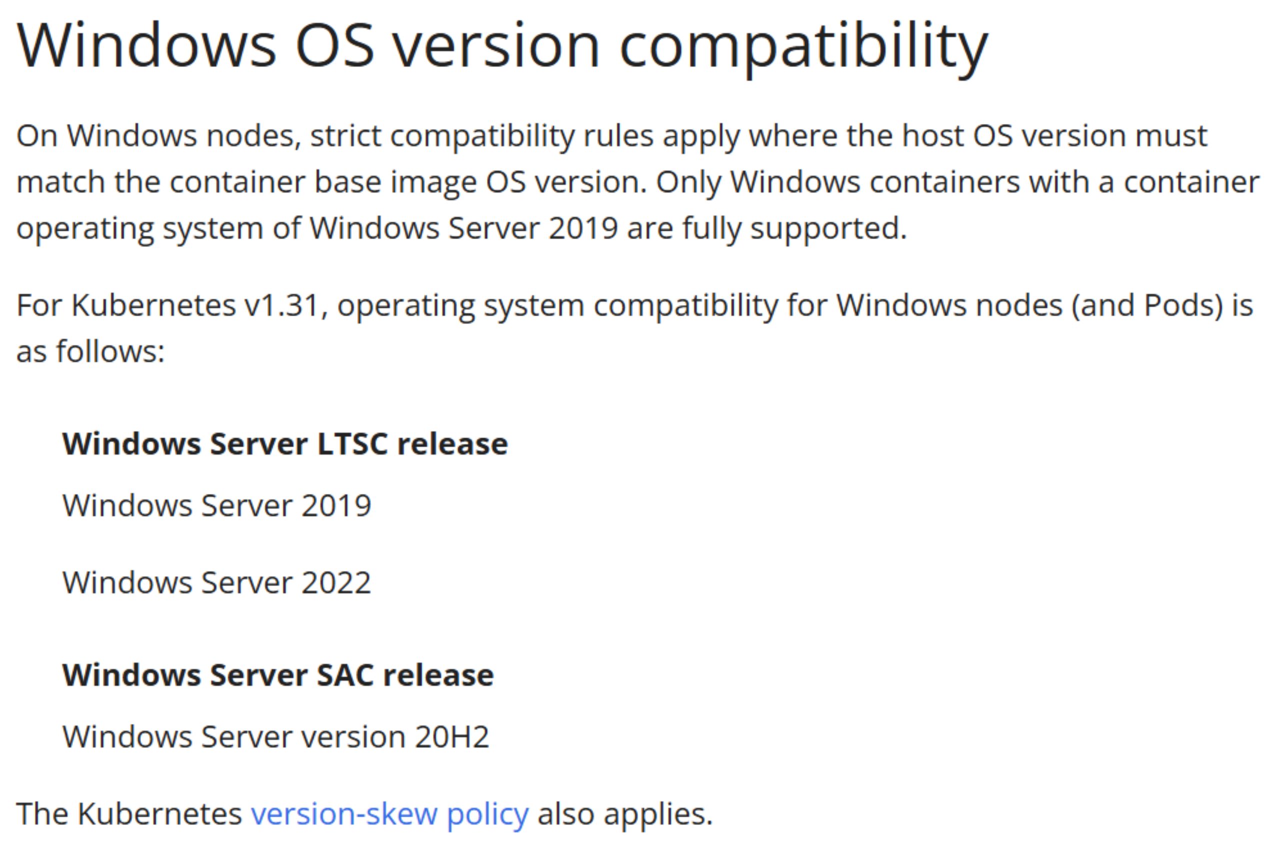
Claim 1	Accused Instrumentalities
	 • **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ○ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ○ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

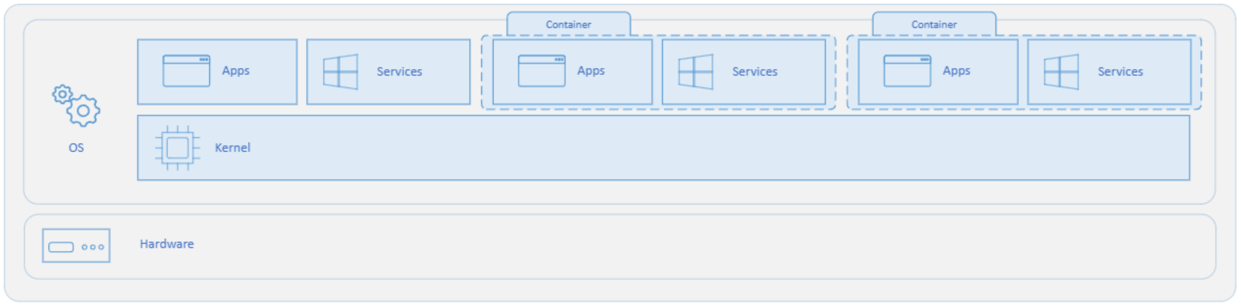
Claim 1	Accused Instrumentalities
[1b] wherein the set of associated system files are compatible with a local kernel of at least some of the plurality of different operating systems,	In the method practiced by Microsoft and/or its customer through the Accused Instrumentalities, the set of associated system files are compatible with a local kernel of at least some of the plurality of different operating systems. The associated system files in the container are compatible with the host kernel, for example because they are linked against the Linux kernel and the supported host operating systems also use the Linux kernel, which has a stable binary interface. In another example, the associated system files are linked against and compatible with a Windows operating system and kernel, and the host operating system runs a compatible Windows operating system with compatible Windows kernel. *See* discussion in element [1a] above. *See also, e.g.*:  **OS** AKS supports Ubuntu 22.04 and Azure Linux 2.0 as the node OS for Linux node pools. For Windows node pools, AKS supports Windows Server 2022 as the default OS. Windows Server 2019 is being retired after Kubernetes version 1.32 reaches end of life and isn't supported in future releases. If you need to upgrade your Windows OS version, see Upgrade from Windows Server 2019 to Windows Server 2022. For more information on using Windows Server on AKS, see Windows container considerations in Azure Kubernetes Service (AKS). https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts  **Container runtime** A container runtime is software that executes containers and manages container images on a node. The runtime helps abstract away sys-calls or OS-specific functionality to run containers on Linux or Windows. For Linux node pools, containerd is used on Kubernetes version 1.19 and higher. For Windows Server 2019 and 2022 node pools, containerd is generally available and is the only runtime option on Kubernetes version 1.23 and higher. https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts

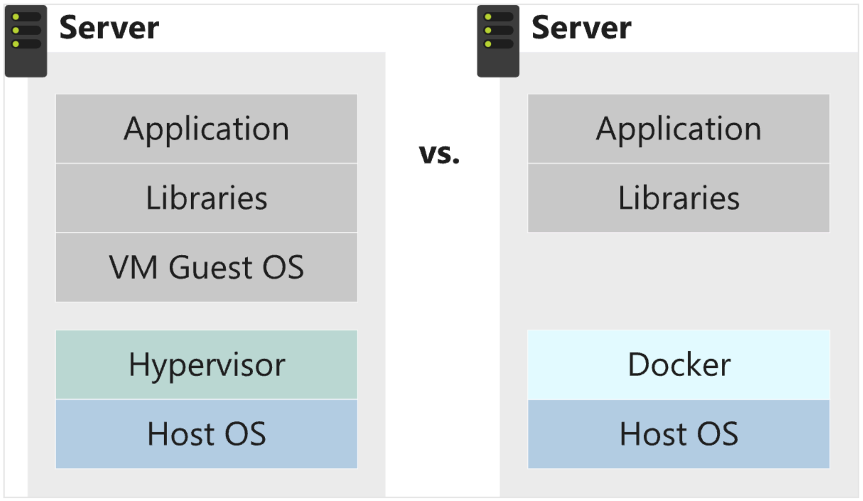
Claim 1	Accused Instrumentalities
	Docker benefits When we use Docker, we immediately get access to the benefits containerization offer. **Efficient hardware use** Containers run without using a virtual machine (VM). As we learned, the container relies on the host kernel for functions such as file system, network management, process scheduling, and memory management.  Server Application Libraries VM Guest OS Hypervisor Host OS vs. Server Application Libraries Docker Host OS Compared to a VM, we can see that a VM requires an OS installed to provide kernel functions to the running applications inside the VM. Keep in mind that the VM OS also requires disk space, memory, and CPU time. By removing the VM and the additional OS requirement, we can free resources on the host and use it for running other containers. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

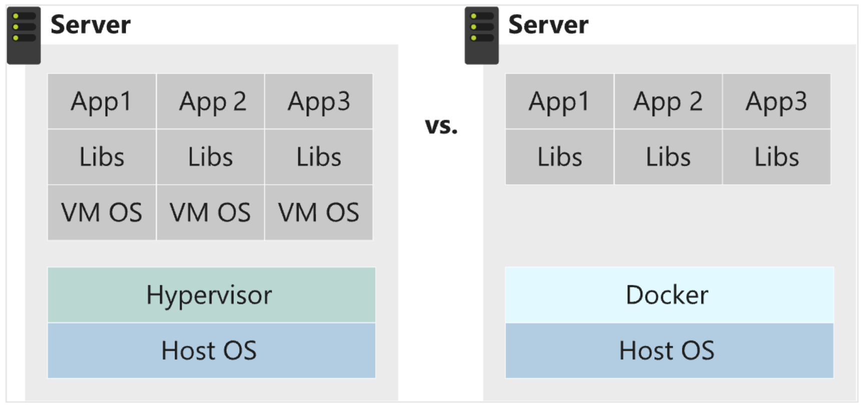
Claim 1	Accused Instrumentalities
	 **Application portability** Containers run almost everywhere: desktops, physical servers, VMs, and in the cloud. This runtime compatibility makes it easy to move containerized applications among different environments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers  **Cloud deployments** Docker containers are the default container architecture the Azure containerization services use, and many other cloud platforms also support them. For instance, you can deploy Docker containers to Azure Container Instances, Azure App Service, and Azure Kubernetes Services. Each of these options provides you with different features and capabilities. For example, Azure container instances allow you to focus on designing and building your applications without the overhead of managing infrastructure. When you have many containers to orchestrate, Azure Kubernetes service makes it easy to deploy and manage large-scale container deployments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

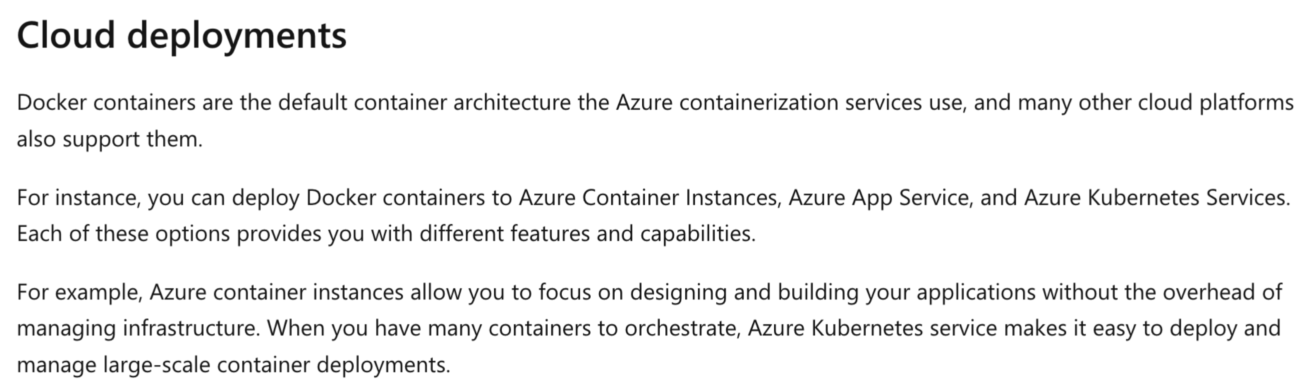
Claim 1	Accused Instrumentalities
	 ## Windows nodes in Kubernetes To enable the orchestration of Windows containers in Kubernetes, include Windows nodes in your existing Linux cluster. Scheduling Windows containers in Pods on Kubernetes is similar to scheduling Linux-based containers. In order to run Windows containers, your Kubernetes cluster must include multiple operating systems. While you can only run the control plane on Linux, you can deploy worker nodes running either Windows or Linux. Windows nodes are supported provided that the operating system is Windows Server 2019 or Windows Server 2022. This document uses the term *Windows containers* to mean Windows containers with process isolation. Kubernetes does not support running Windows containers with Hyper-V isolation. https://kubernetes.io/docs/concepts/windows/intro/

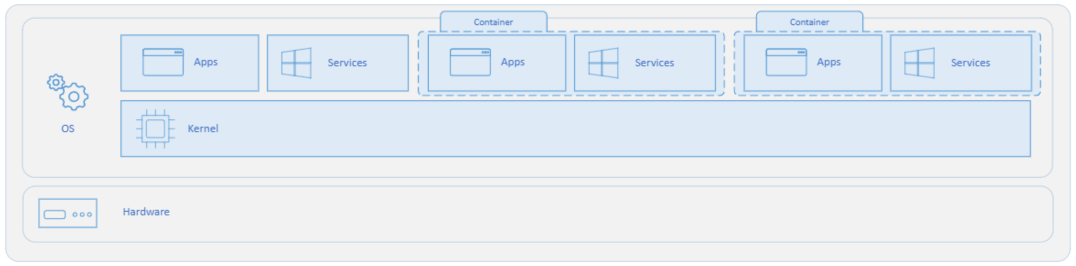
Claim 1	Accused Instrumentalities
	 ## Windows OS version compatibility On Windows nodes, strict compatibility rules apply where the host OS version must match the container base image OS version. Only Windows containers with a container operating system of Windows Server 2019 are fully supported. For Kubernetes v1.31, operating system compatibility for Windows nodes (and Pods) is as follows: **Windows Server LTSC release** Windows Server 2019 Windows Server 2022 **Windows Server SAC release** Windows Server version 20H2 The Kubernetes version-skew policy also applies. https://kubernetes.io/docs/concepts/windows/intro/#windows-os-version-support

Claim 1	Accused Instrumentalities
	How containers work A container is an isolated, lightweight package for running an application on the host operating system. Containers build on top of the host operating system's kernel (which can be thought of as the buried plumbing of the operating system), as shown in the diagram below.  Container Container Apps Services Apps Services Apps Services OS Kernel Hardware While a container shares the host operating system's kernel, the container doesn't get unfettered access to it. Instead, the container gets an isolated–and in some cases virtualized–view of the system. For example, a container can access a virtualized version of the file system and registry, but any changes affect only the container and are discarded when it stops. To save data, the container can mount persistent storage such as an Azure Disk or a file share (including Azure Files). A container builds on top of the kernel, but the kernel doesn't provide all of the APIs and services an app needs to run–most of these are provided by system files (libraries) that run above the kernel in user mode. Because a container is isolated from the host's user mode environment, the container needs its own copy of these user mode system files, which are packaged into something known as a base image. The base image serves as the foundational layer upon which your container is built, providing it with operating system services not provided by the kernel. But we'll talk more about container images later. https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

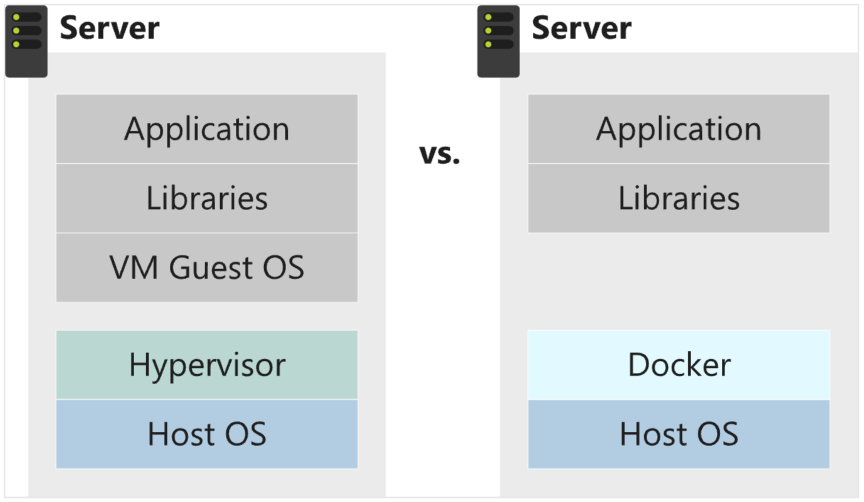
Claim 1	Accused Instrumentalities
[1c] the containers of application software excluding a kernel,	In the method practiced by Microsoft and/or its customer through the Accused Instrumentalities, the containers of application software exclude a kernel. *See, e.g.*: **Docker benefits** When we use Docker, we immediately get access to the benefits containerization offer. **Efficient hardware use** Containers run without using a virtual machine (VM). As we learned, the container relies on the host kernel for functions such as file system, network management, process scheduling, and memory management.  Server Application Libraries VM Guest OS Hypervisor Host OS vs. Server Application Libraries Docker Host OS Compared to a VM, we can see that a VM requires an OS installed to provide kernel functions to the running applications inside the VM. Keep in mind that the VM OS also requires disk space, memory, and CPU time. By removing the VM and the additional OS requirement, we can free resources on the host and use it for running other containers. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

Claim 1	Accused Instrumentalities
	Container isolation Docker containers provide security features to run multiple containers simultaneously on the same host without affecting each other. As we learned, we can configure both data storage and network configuration to isolate our containers or share data and connectivity between specific containers. Let's compare this feature to using VMs.  Server App1 App 2 App3 Libs Libs Libs VM OS VM OS VM OS Hypervisor Host OS vs. Server App1 App 2 App3 Libs Libs Libs Docker Host OS Assume we have a physical host running two VMs. We have three applications that we want to run isolated from each other. We decide to deploy the first app onto VM1 and the second onto VM2 to separate the two apps from each other. If we now choose to install the third application, we'll need to install another VM to continue this pattern. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers **Application portability** Containers run almost everywhere: desktops, physical servers, VMs, and in the cloud. This runtime compatibility makes it easy to move containerized applications among different environments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

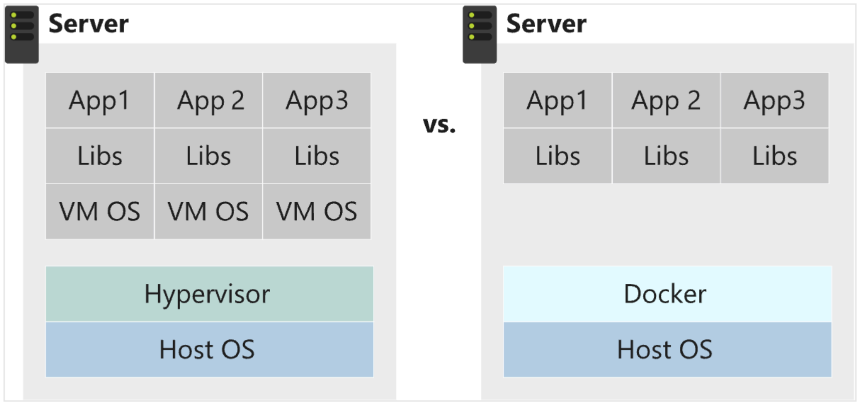
Claim 1	Accused Instrumentalities
	 **Cloud deployments** Docker containers are the default container architecture the Azure containerization services use, and many other cloud platforms also support them. For instance, you can deploy Docker containers to Azure Container Instances, Azure App Service, and Azure Kubernetes Services. Each of these options provides you with different features and capabilities. For example, Azure container instances allow you to focus on designing and building your applications without the overhead of managing infrastructure. When you have many containers to orchestrate, Azure Kubernetes service makes it easy to deploy and manage large-scale container deployments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

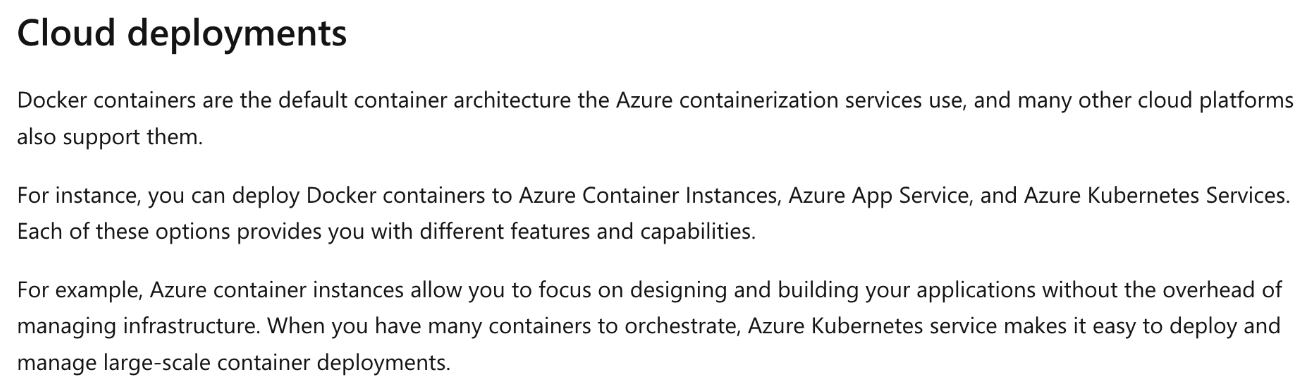
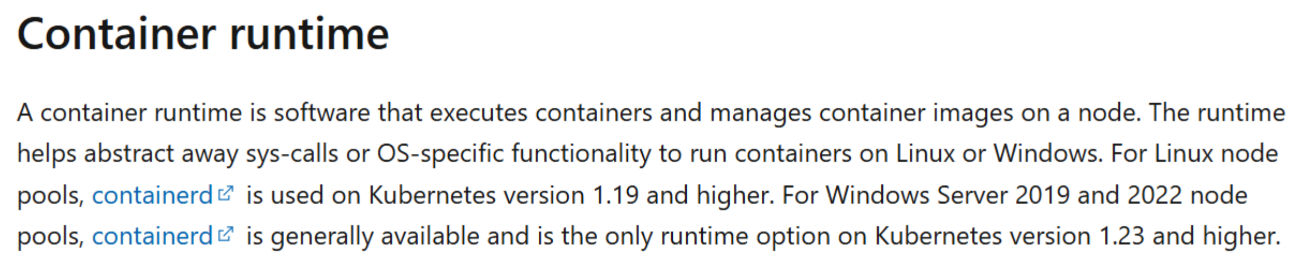
Claim 1	Accused Instrumentalities
	## How containers work A container is an isolated, lightweight package for running an application on the host operating system. Containers build on top of the host operating system's kernel (which can be thought of as the buried plumbing of the operating system), as shown in the diagram below.  Container Container Apps Services Apps Services Apps Services OS Kernel Hardware While a container shares the host operating system's kernel, the container doesn't get unfettered access to it. Instead, the container gets an isolated–and in some cases virtualized–view of the system. For example, a container can access a virtualized version of the file system and registry, but any changes affect only the container and are discarded when it stops. To save data, the container can mount persistent storage such as an Azure Disk or a file share (including Azure Files). A container builds on top of the kernel, but the kernel doesn't provide all of the APIs and services an app needs to run–most of these are provided by system files (libraries) that run above the kernel in user mode. Because a container is isolated from the host's user mode environment, the container needs its own copy of these user mode system files, which are packaged into something known as a base image. The base image serves as the foundational layer upon which your container is built, providing it with operating system services not provided by the kernel. But we'll talk more about container images later. https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/ 6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/

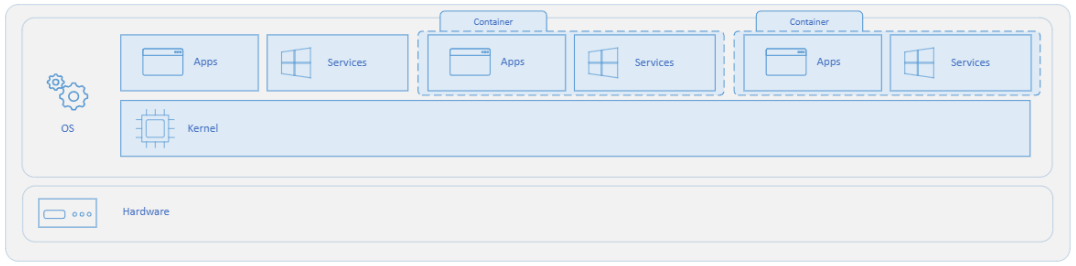
Claim 1	Accused Instrumentalities
[1d] wherein some or all of the associated system files within a container stored in memory are utilized in place of the associated local system files that remain resident on the server,	In the method practiced by Microsoft and/or its customer through the Accused Instrumentalities, some or all of the associated system files within a container stored in memory are utilized in place of the associated local system files that remain resident on the server. For example, each container will utilize its own associated system files, including runtime linked libraries (*e.g.* .dll/.so files), configuration files, etc., not the corresponding associated local system files (*e.g.*, libraries and configuration files of the host OS). As described above and below, in the Accused Instrumentalities the associated system files provide at least some of the same functionalities as the associated local system files. The host/node's associated local system files remain resident on the host/node, for example for use by system processes or applications outside the container environment. *See, e.g.*:

Claim 1	Accused Instrumentalities
	## Docker benefits When we use Docker, we immediately get access to the benefits containerization offer. ### Efficient hardware use Containers run without using a virtual machine (VM). As we learned, the container relies on the host kernel for functions such as file system, network management, process scheduling, and memory management.  Server Application Libraries VM Guest OS Hypervisor Host OS vs. Server Application Libraries Docker Host OS Compared to a VM, we can see that a VM requires an OS installed to provide kernel functions to the running applications inside the VM. Keep in mind that the VM OS also requires disk space, memory, and CPU time. By removing the VM and the additional OS requirement, we can free resources on the host and use it for running other containers. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

Case 5:24-cv-04740-PCP Document 63-2 Filed 12/20/25 Page 48 of 72

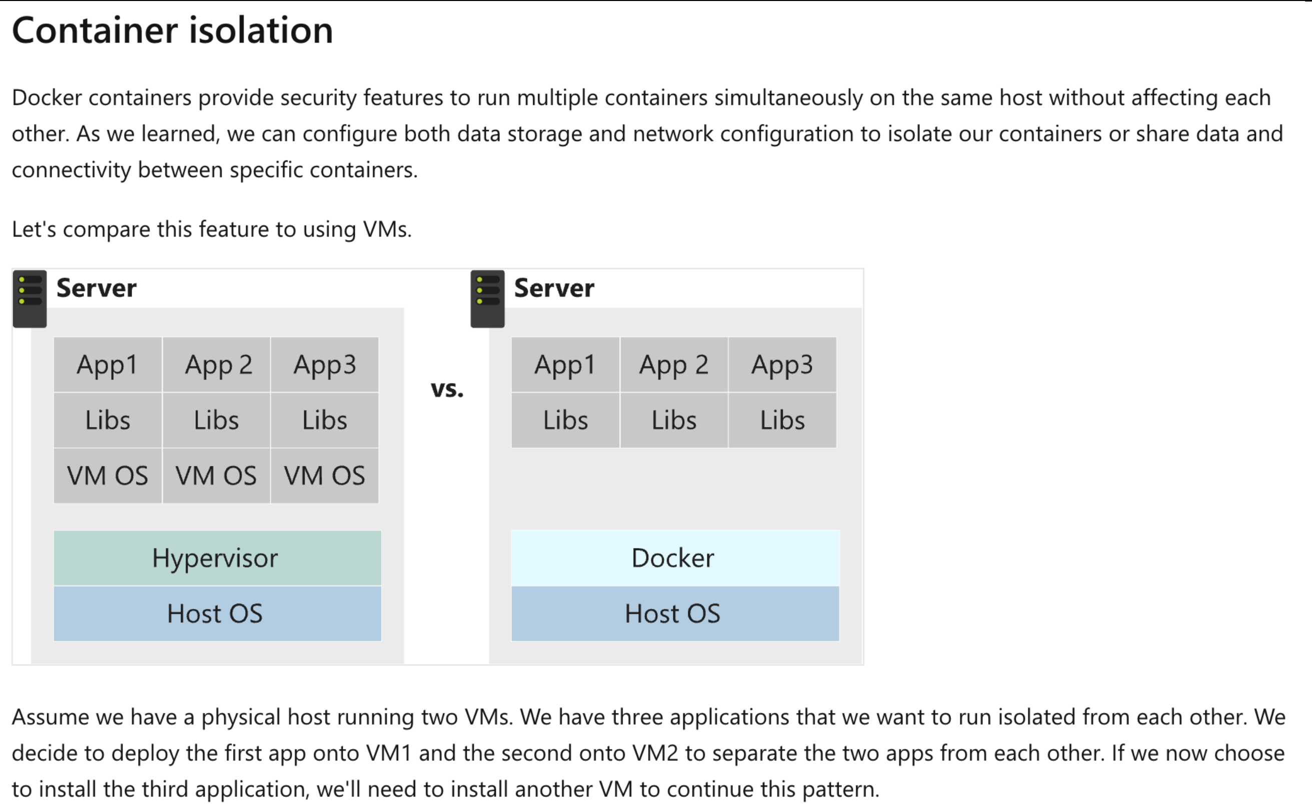
Claim 1	Accused Instrumentalities
	Container isolation Docker containers provide security features to run multiple containers simultaneously on the same host without affecting each other. As we learned, we can configure both data storage and network configuration to isolate our containers or share data and connectivity between specific containers. Let's compare this feature to using VMs.  Server App1 App 2 App3 Libs Libs Libs VM OS VM OS VM OS Hypervisor Host OS vs. Server App1 App 2 App3 Libs Libs Libs Docker Host OS Assume we have a physical host running two VMs. We have three applications that we want to run isolated from each other. We decide to deploy the first app onto VM1 and the second onto VM2 to separate the two apps from each other. If we now choose to install the third application, we'll need to install another VM to continue this pattern. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers **Application portability** Containers run almost everywhere: desktops, physical servers, VMs, and in the cloud. This runtime compatibility makes it easy to move containerized applications among different environments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

Claim 1	Accused Instrumentalities
	 ## Cloud deployments Docker containers are the default container architecture the Azure containerization services use, and many other cloud platforms also support them. For instance, you can deploy Docker containers to Azure Container Instances, Azure App Service, and Azure Kubernetes Services. Each of these options provides you with different features and capabilities. For example, Azure container instances allow you to focus on designing and building your applications without the overhead of managing infrastructure. When you have many containers to orchestrate, Azure Kubernetes service makes it easy to deploy and manage large-scale container deployments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers  ## Container runtime A container runtime is software that executes containers and manages container images on a node. The runtime helps abstract away sys-calls or OS-specific functionality to run containers on Linux or Windows. For Linux node pools, containerd is used on Kubernetes version 1.19 and higher. For Windows Server 2019 and 2022 node pools, containerd is generally available and is the only runtime option on Kubernetes version 1.23 and higher. https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts

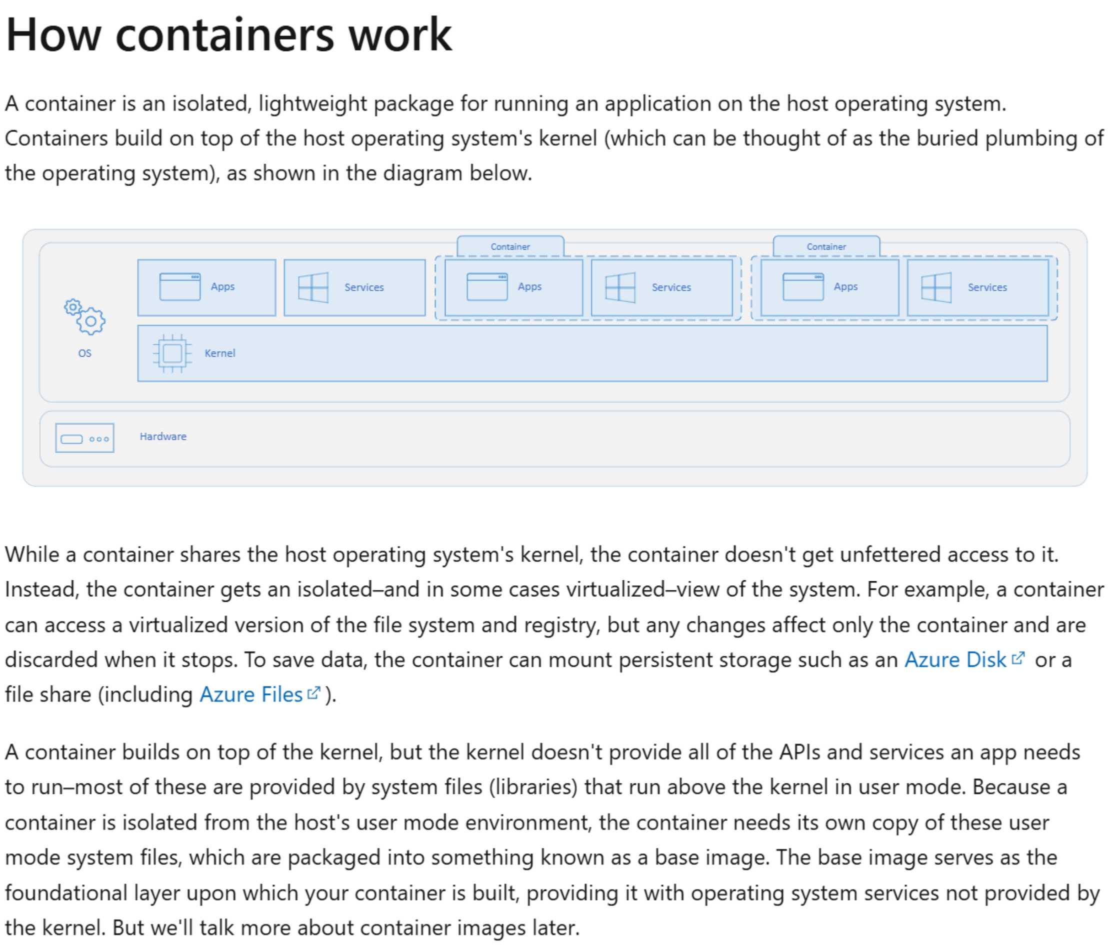
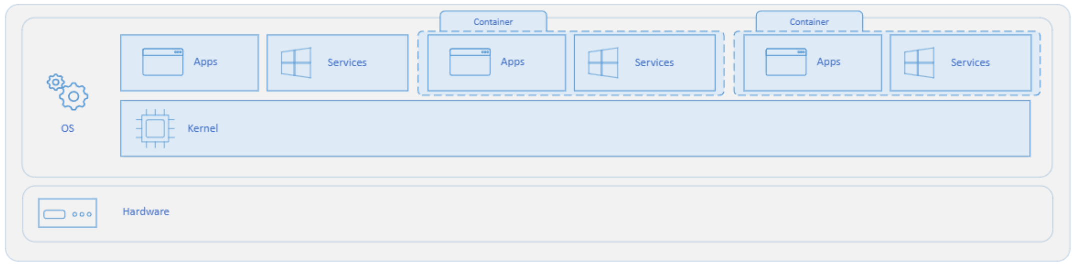
Claim 1	Accused Instrumentalities
	How containers work A container is an isolated, lightweight package for running an application on the host operating system. Containers build on top of the host operating system's kernel (which can be thought of as the buried plumbing of the operating system), as shown in the diagram below.  Container Container OS Apps Services Apps Services Apps Services Kernel Hardware While a container shares the host operating system's kernel, the container doesn't get unfettered access to it. Instead, the container gets an isolated–and in some cases virtualized–view of the system. For example, a container can access a virtualized version of the file system and registry, but any changes affect only the container and are discarded when it stops. To save data, the container can mount persistent storage such as an Azure Disk or a file share (including Azure Files). A container builds on top of the kernel, but the kernel doesn't provide all of the APIs and services an app needs to run–most of these are provided by system files (libraries) that run above the kernel in user mode. Because a container is isolated from the host's user mode environment, the container needs its own copy of these user mode system files, which are packaged into something known as a base image. The base image serves as the foundational layer upon which your container is built, providing it with operating system services not provided by the kernel. But we'll talk more about container images later. https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/ 6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/

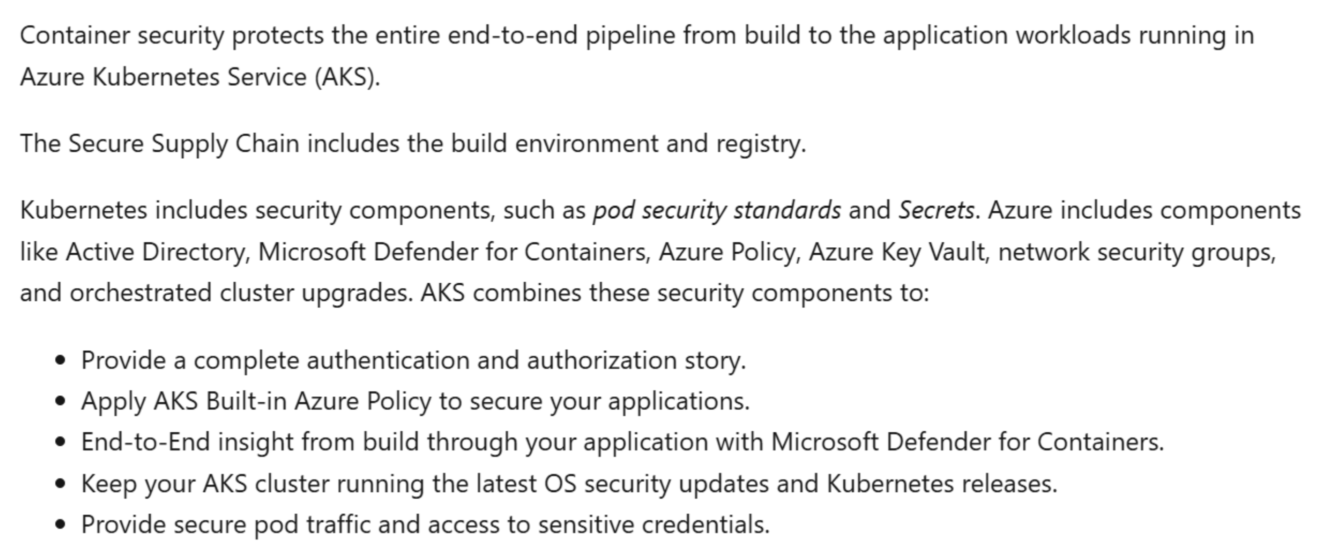
Claim 1	Accused Instrumentalities
[1e] wherein said associated system files utilized in place of the associated local system files are copies or modified copies of the associated local system files that remain resident on the server,	In the method practiced by Microsoft and/or its customer through the Accused Instrumentalities, said associated system files utilized in place of the associated local system files are copies or modified copies of the associated local system files that remain resident on the server. For example, in some cases the host OS and container will use one or more identical system files, for example when both the host and the container incorporate the same Linux distribution version, or when both host and container use the same Windows services. In the case where the associated system files are identical to the associated local system files, they are copies thereof. In other cases modified copies are used instead, for example when different versions of the same library, or configuration files with different parameters, are used by the host and container. *See* discussion in elements [1a], [1b] above.

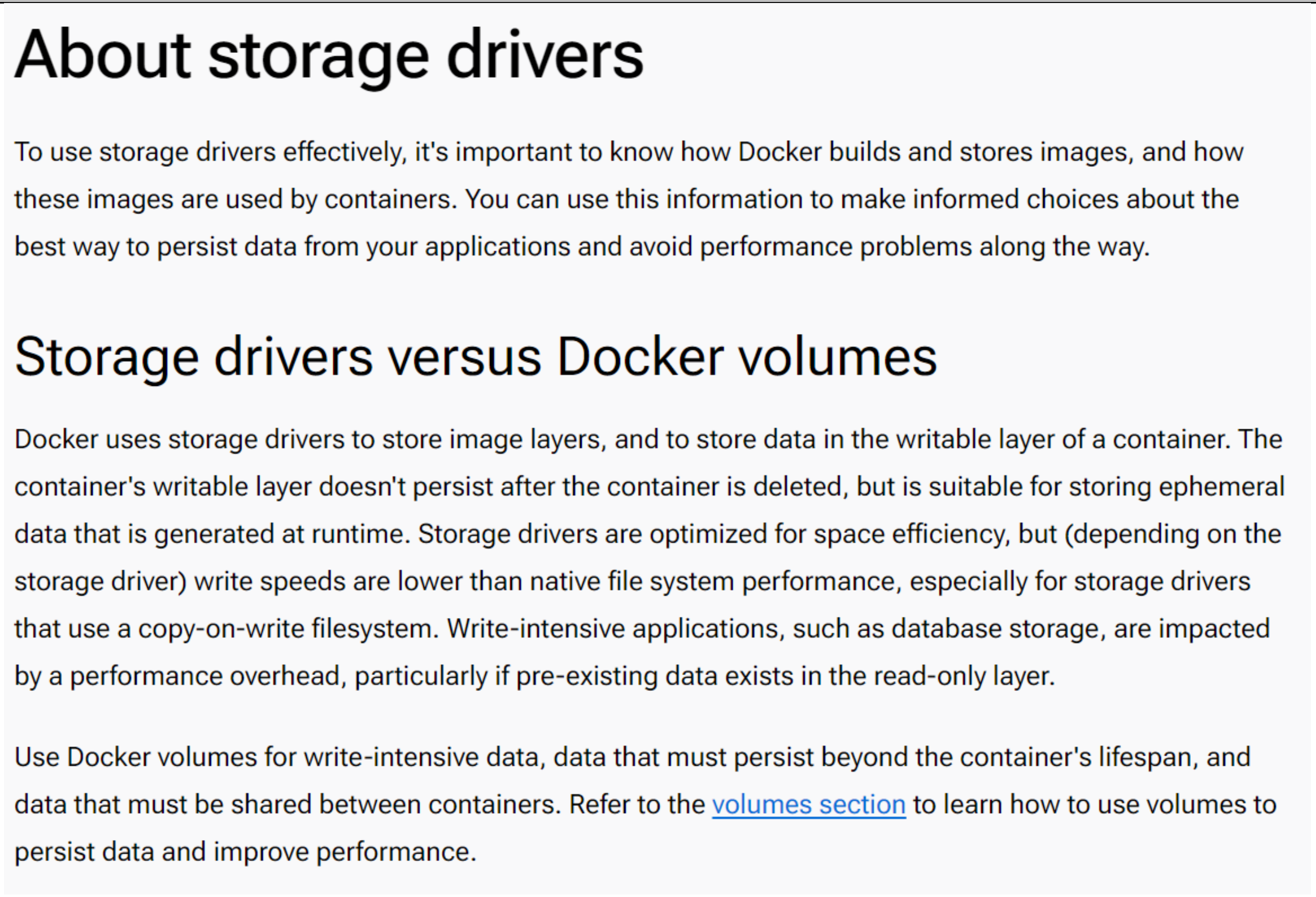
Claim 1	Accused Instrumentalities
[1f] and wherein the application software cannot be shared between the plurality of secure containers of application software,	In the method practiced by Microsoft and/or its customer through the Accused Instrumentalities, the application software cannot be shared between the plurality of secure containers of application software. For example, each container has an isolated runtime environment that cannot be accessed by other containers, for example including a per-container writeable layer or other ephemeral per-container storage. For another example, when the plurality of secure containers each corresponds to a different container image, each container cannot access another container’s image and therefore application software. *See, e.g.*:

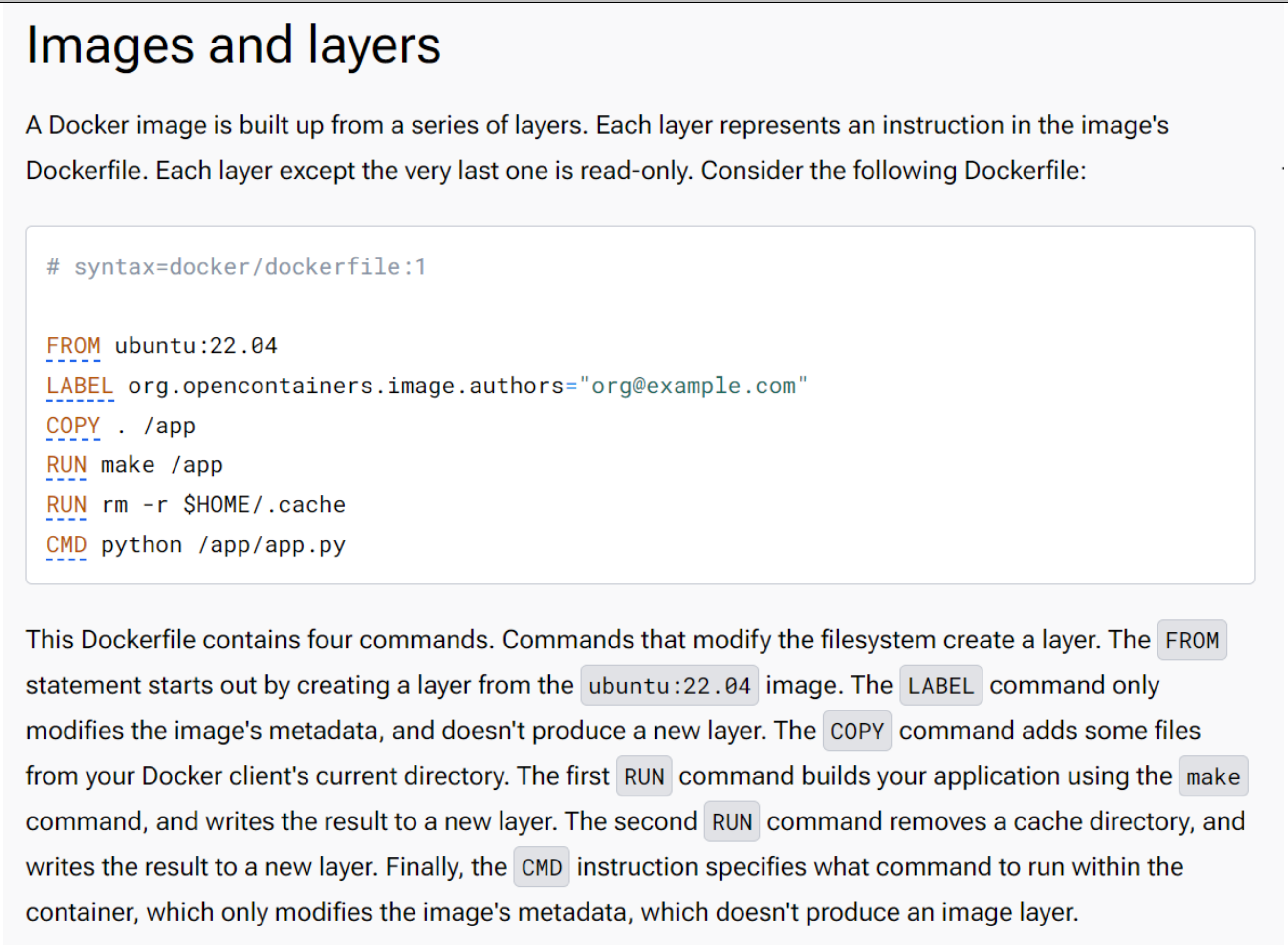
Claim 1	Accused Instrumentalities
	 **Container isolation** Docker containers provide security features to run multiple containers simultaneously on the same host without affecting each other. As we learned, we can configure both data storage and network configuration to isolate our containers or share data and connectivity between specific containers. Let's compare this feature to using VMs. Assume we have a physical host running two VMs. We have three applications that we want to run isolated from each other. We decide to deploy the first app onto VM1 and the second onto VM2 to separate the two apps from each other. If we now choose to install the third application, we'll need to install another VM to continue this pattern. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

Case 4:24-cv-04703-PCP Document 1-3 Filed 12/20/24 Page 54 of 72

Claim 1	Accused Instrumentalities
	 **How containers work** A container is an isolated, lightweight package for running an application on the host operating system. Containers build on top of the host operating system's kernel (which can be thought of as the buried plumbing of the operating system), as shown in the diagram below.  While a container shares the host operating system's kernel, the container doesn't get unfettered access to it. Instead, the container gets an isolated–and in some cases virtualized–view of the system. For example, a container can access a virtualized version of the file system and registry, but any changes affect only the container and are discarded when it stops. To save data, the container can mount persistent storage such as an Azure Disk or a file share (including Azure Files). A container builds on top of the kernel, but the kernel doesn't provide all of the APIs and services an app needs to run–most of these are provided by system files (libraries) that run above the kernel in user mode. Because a container is isolated from the host's user mode environment, the container needs its own copy of these user mode system files, which are packaged into something known as a base image. The base image serves as the foundational layer upon which your container is built, providing it with operating system services not provided by the kernel. But we'll talk more about container images later. https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

Claim 1	Accused Instrumentalities
	 Container security protects the entire end-to-end pipeline from build to the application workloads running in Azure Kubernetes Service (AKS). The Secure Supply Chain includes the build environment and registry. Kubernetes includes security components, such as *pod security standards* and *Secrets*. Azure includes components like Active Directory, Microsoft Defender for Containers, Azure Policy, Azure Key Vault, network security groups, and orchestrated cluster upgrades. AKS combines these security components to: • Provide a complete authentication and authorization story. • Apply AKS Built-in Azure Policy to secure your applications. • End-to-End insight from build through your application with Microsoft Defender for Containers. • Keep your AKS cluster running the latest OS security updates and Kubernetes releases. • Provide secure pod traffic and access to sensitive credentials. https://learn.microsoft.com/en-us/azure/aks/concepts-security

Claim 1	Accused Instrumentalities
	 **About storage drivers** To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	 https://docs.docker.com/storage/storagedriver/

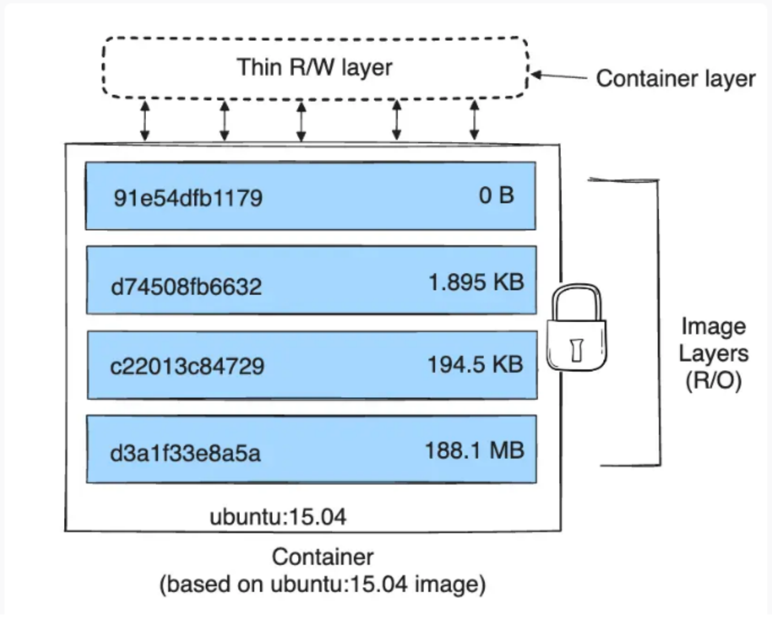
Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

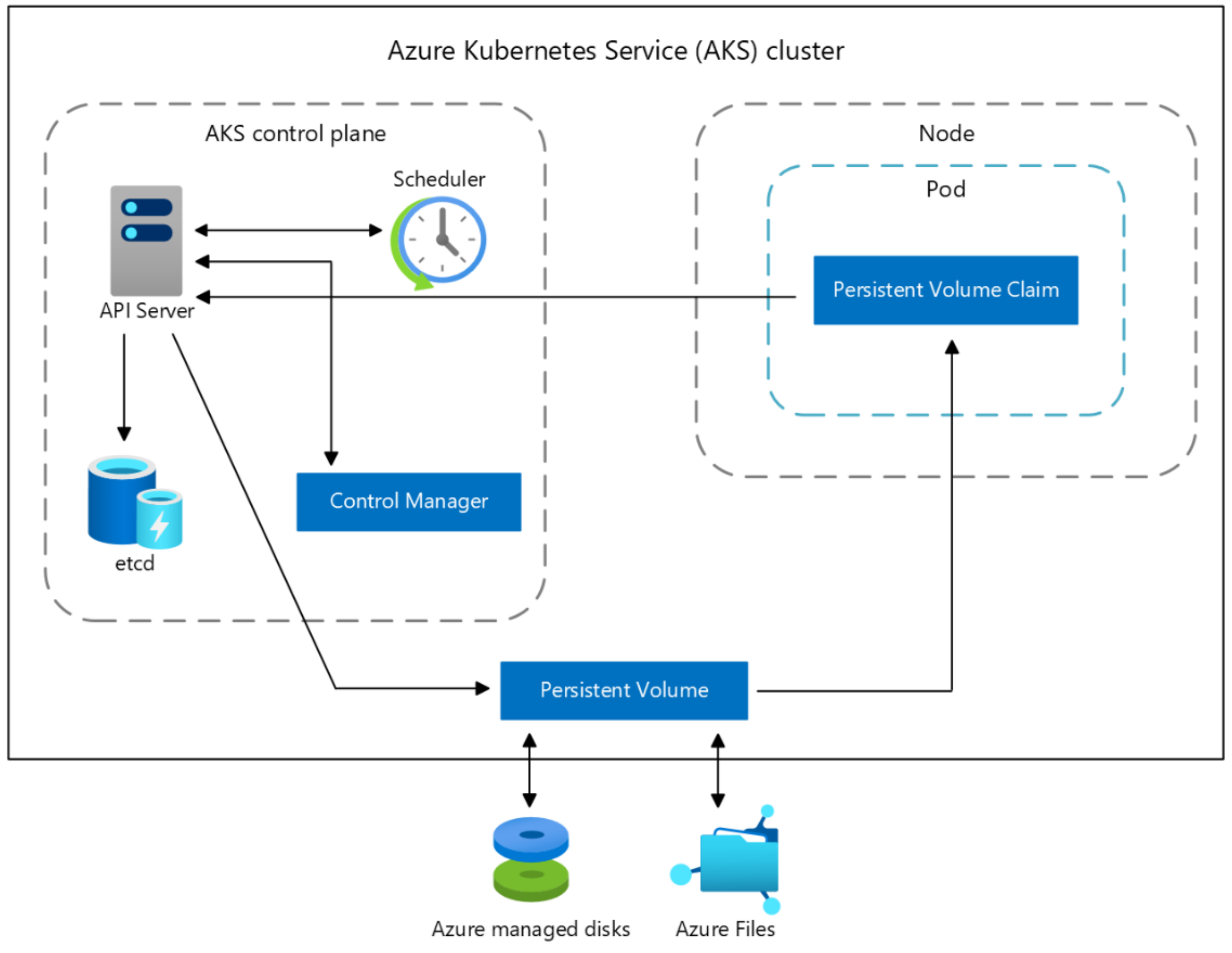
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

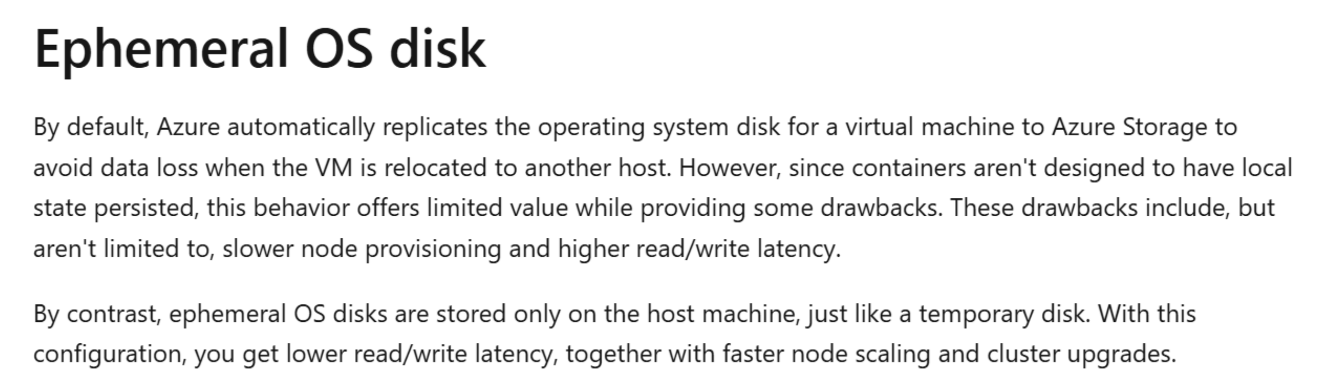
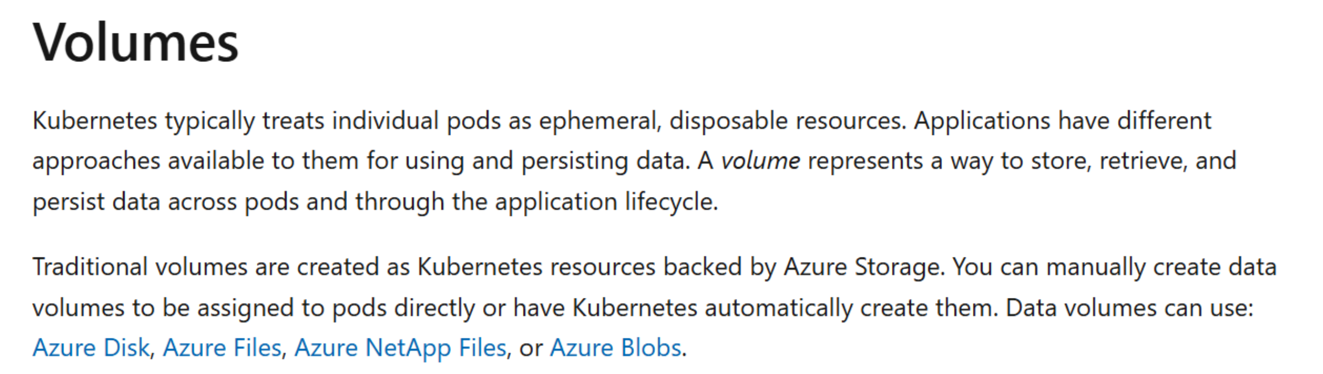
This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

Case 4:24-04740-BCP Document 1-3-2 Filed 12/02/25 Page 58 of 72

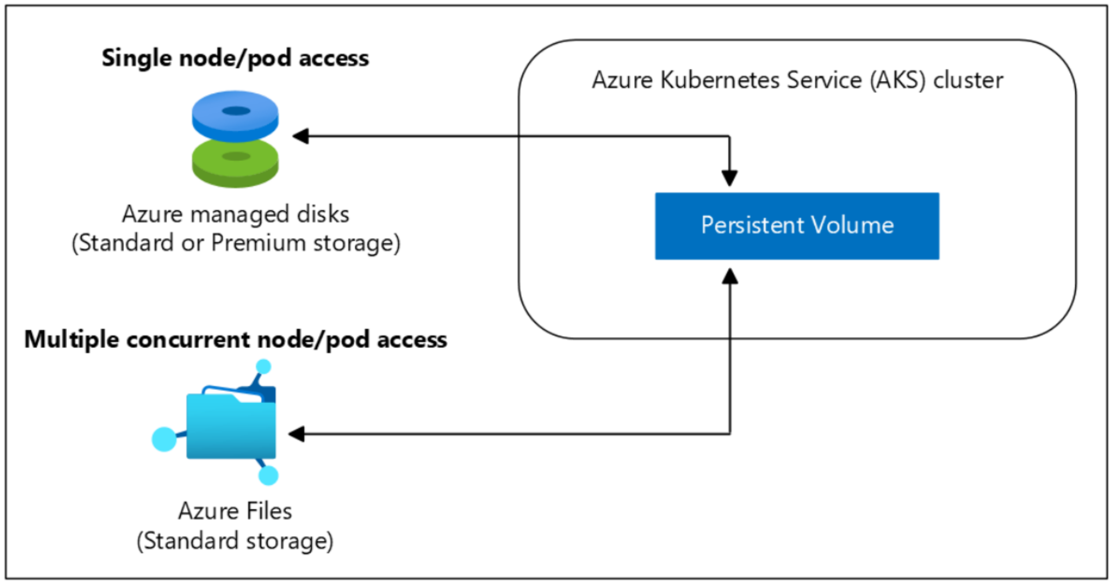
Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB Image Layers (R/O) ubuntu:15.04 Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

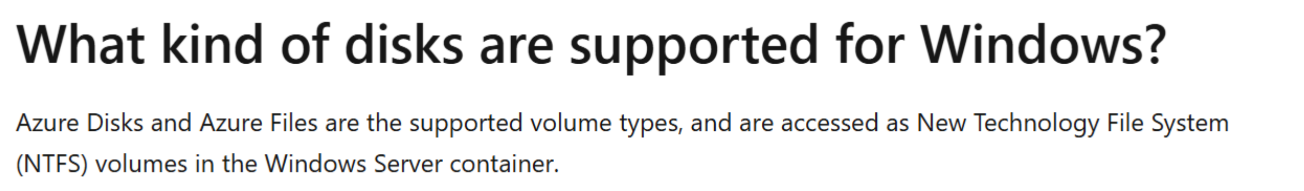
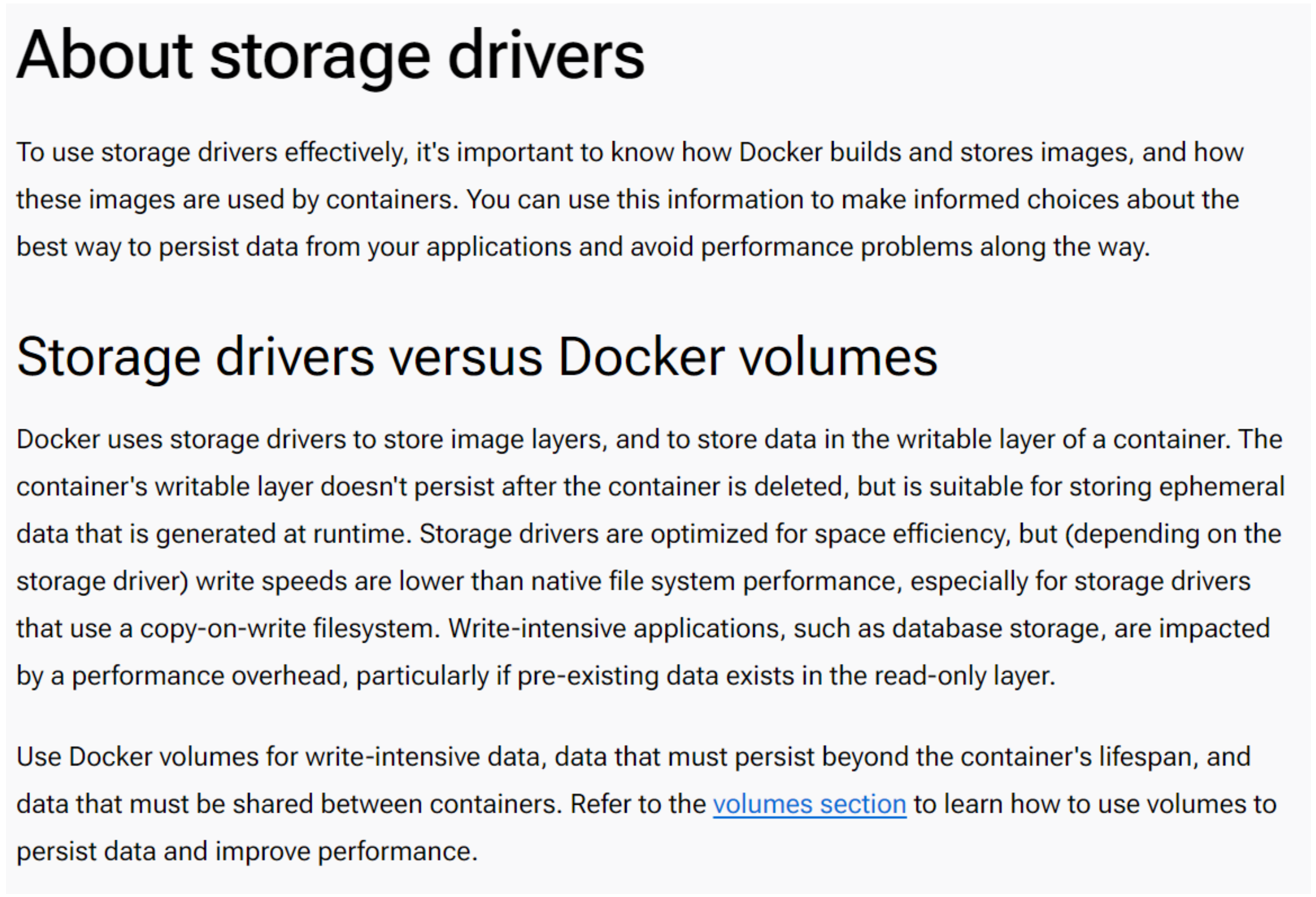
Claim 1	Accused Instrumentalities
[1g] and wherein each of the containers has a unique root file system that is different from an operating system's root file system.	In the method practiced by Microsoft and/or its customer through the Accused Instrumentalities, each of the containers has a unique root file system that is different from an operating system's root file system. For example, the container's root file system comprises the image layer(s), an ephemeral writeable layer (e.g., in Docker terminology the container layer), and optionally one or more volumes. This root file system is distinct and isolated from the host operating system's root file system. *See, e.g.*:

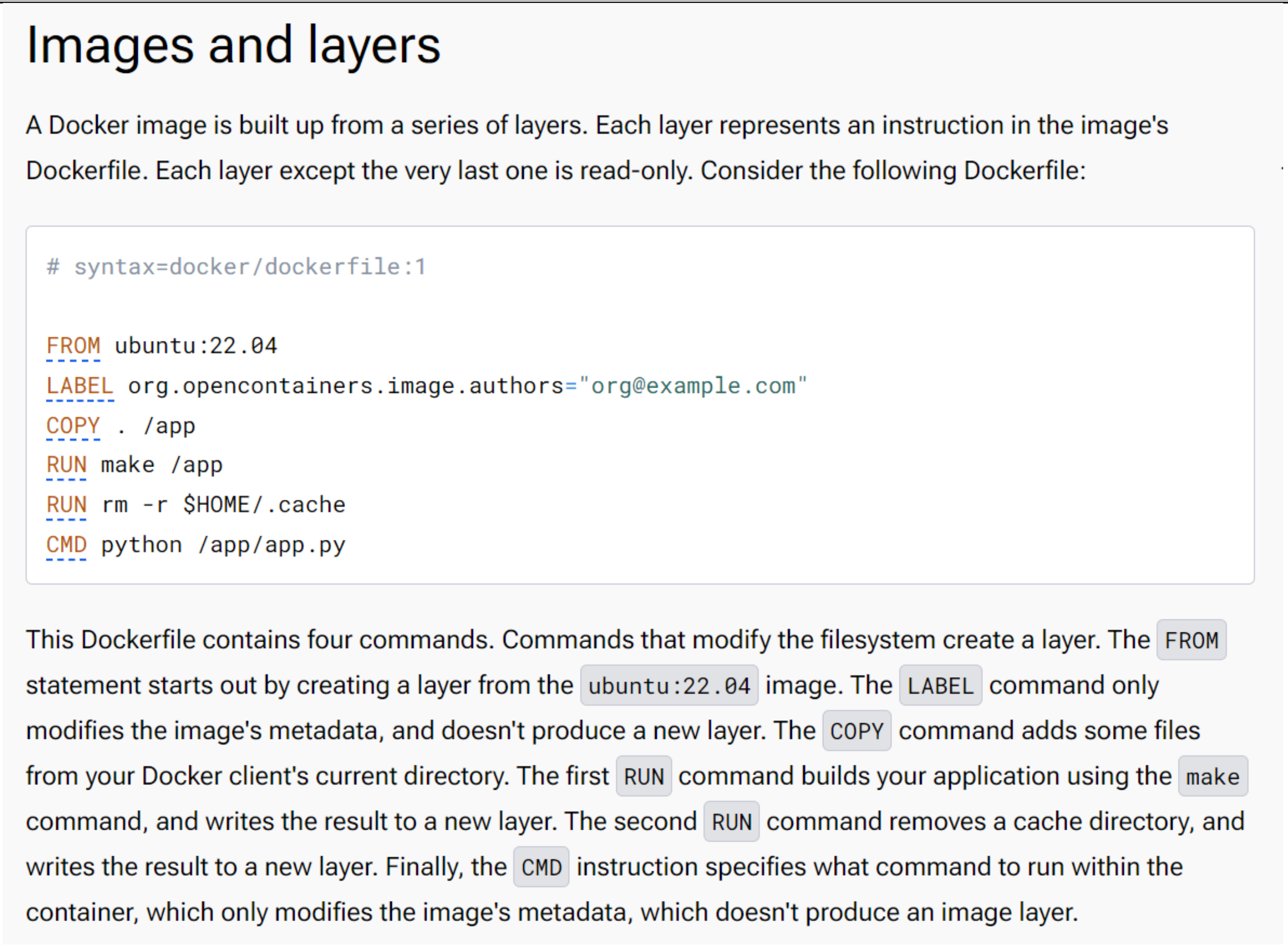
Claim 1	Accused Instrumentalities
	 Azure Kubernetes Service (AKS) cluster AKS control plane Scheduler API Server etcd Control Manager Node Pod Persistent Volume Claim Persistent Volume Azure managed disks Azure Files https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Claim 1	Accused Instrumentalities
	 **Ephemeral OS disk** By default, Azure automatically replicates the operating system disk for a virtual machine to Azure Storage to avoid data loss when the VM is relocated to another host. However, since containers aren't designed to have local state persisted, this behavior offers limited value while providing some drawbacks. These drawbacks include, but aren't limited to, slower node provisioning and higher read/write latency. By contrast, ephemeral OS disks are stored only on the host machine, just like a temporary disk. With this configuration, you get lower read/write latency, together with faster node scaling and cluster upgrades. https://learn.microsoft.com/en-us/azure/aks/concepts-storage  **Volumes** Kubernetes typically treats individual pods as ephemeral, disposable resources. Applications have different approaches available to them for using and persisting data. A *volume* represents a way to store, retrieve, and persist data across pods and through the application lifecycle. Traditional volumes are created as Kubernetes resources backed by Azure Storage. You can manually create data volumes to be assigned to pods directly or have Kubernetes automatically create them. Data volumes can use: Azure Disk, Azure Files, Azure NetApp Files, or Azure Blobs. https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Case 5:24-cv-04740-PCP Document 1-3 Filed 12/20/24 Page 62 of 72

Claim 1	Accused Instrumentalities
	Persistent volumes Volumes defined and created as part of the pod lifecycle only exist until you delete the pod. Pods often expect their storage to remain if a pod is rescheduled on a different host during a maintenance event, especially in StatefulSets. A *persistent volume* (PV) is a storage resource created and managed by the Kubernetes API that can exist beyond the lifetime of an individual pod. You can use the following Azure Storage services to provide the persistent volume: • Azure Disk • Azure Files • Azure Container Storage As noted in the Volumes section, the choice of Azure Disks or Azure Files is often determined by the need for concurrent access to the data or the performance tier.  Single node/pod access Azure managed disks (Standard or Premium storage) Multiple concurrent node/pod access Azure Files (Standard storage) Azure Kubernetes Service (AKS) cluster Persistent Volume https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Claim 1	Accused Instrumentalities
	 **What kind of disks are supported for Windows?** Azure Disks and Azure Files are the supported volume types, and are accessed as New Technology File System (NTFS) volumes in the Windows Server container. https://learn.microsoft.com/en-us/azure/aks/windows-faq  **About storage drivers** To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

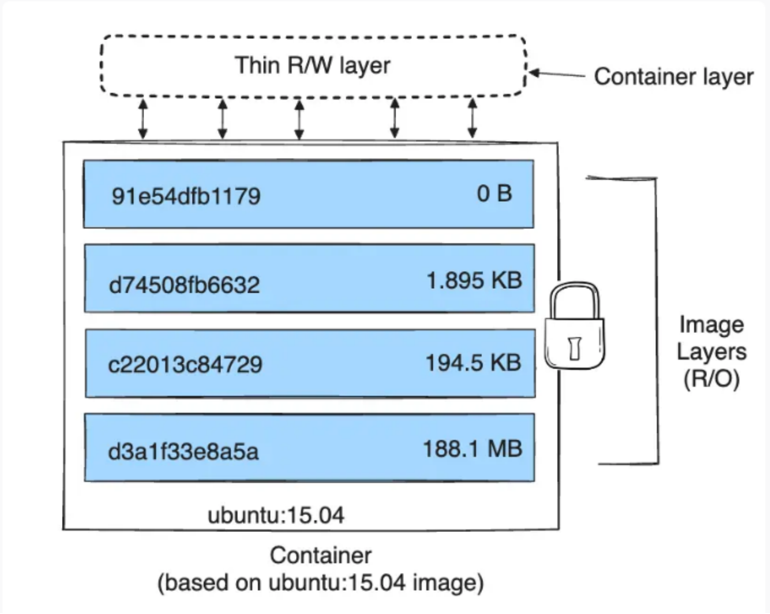
Claim 1	Accused Instrumentalities
	 https://docs.docker.com/storage/storagedriver/

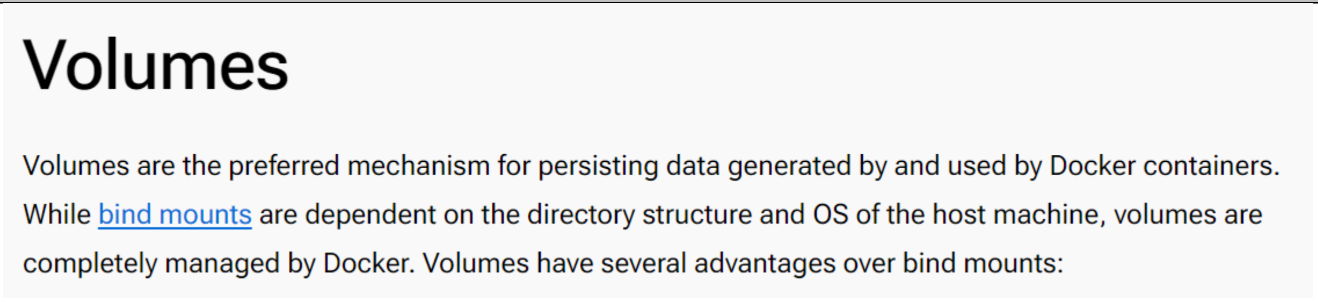
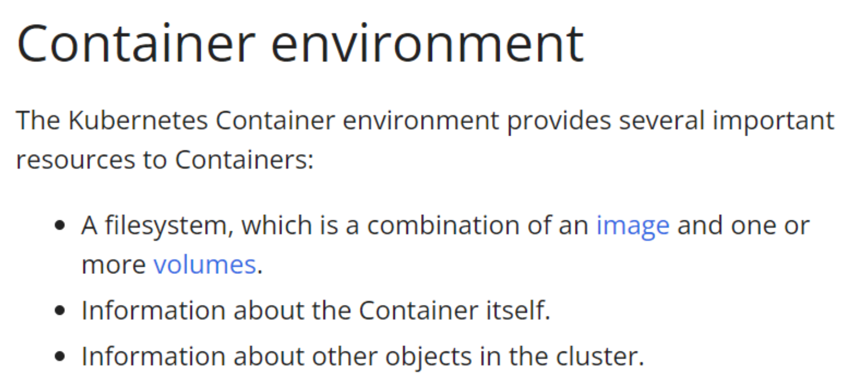
Images and layers

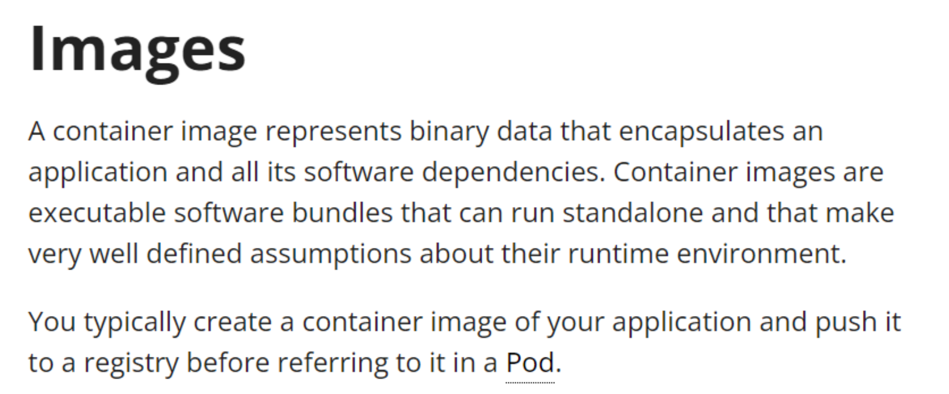
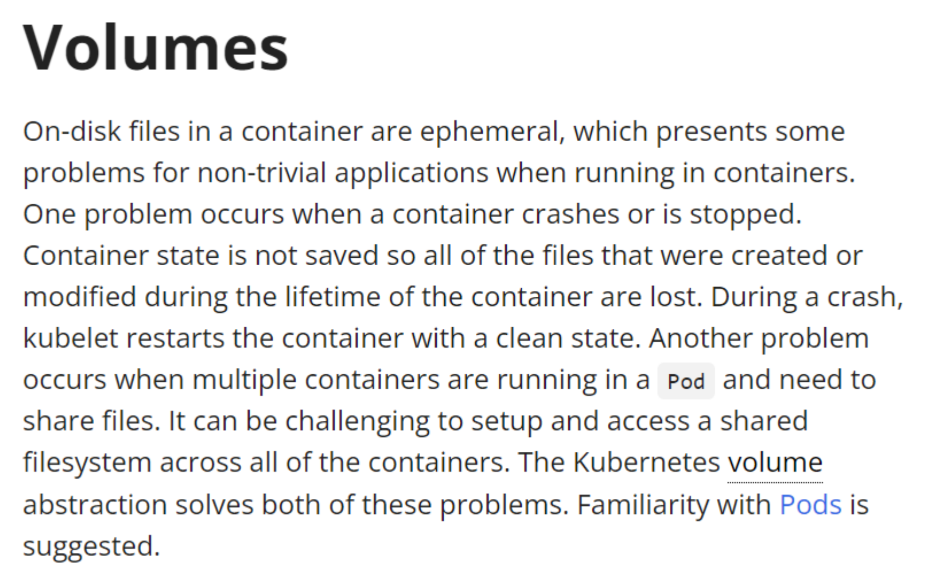
A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

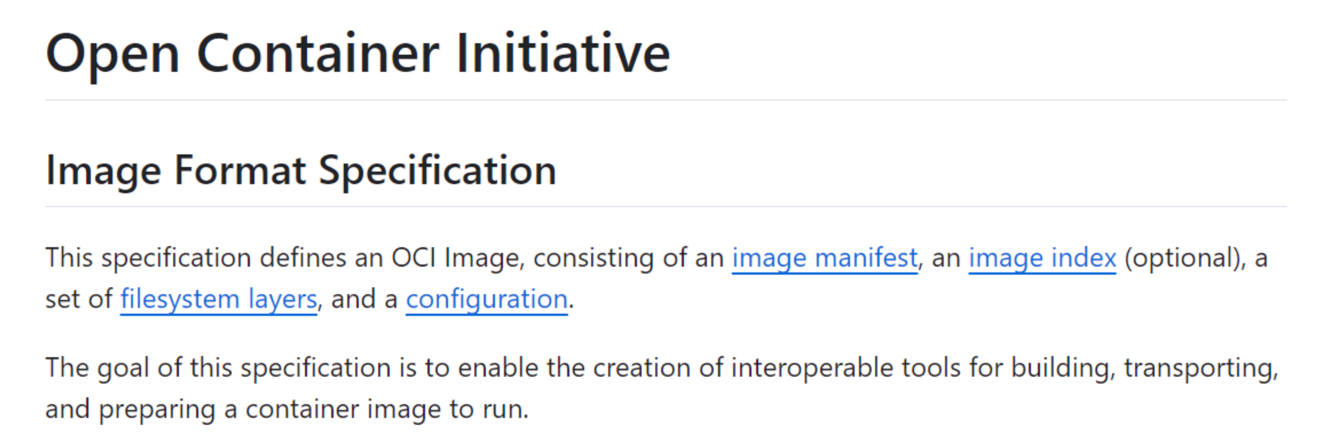
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

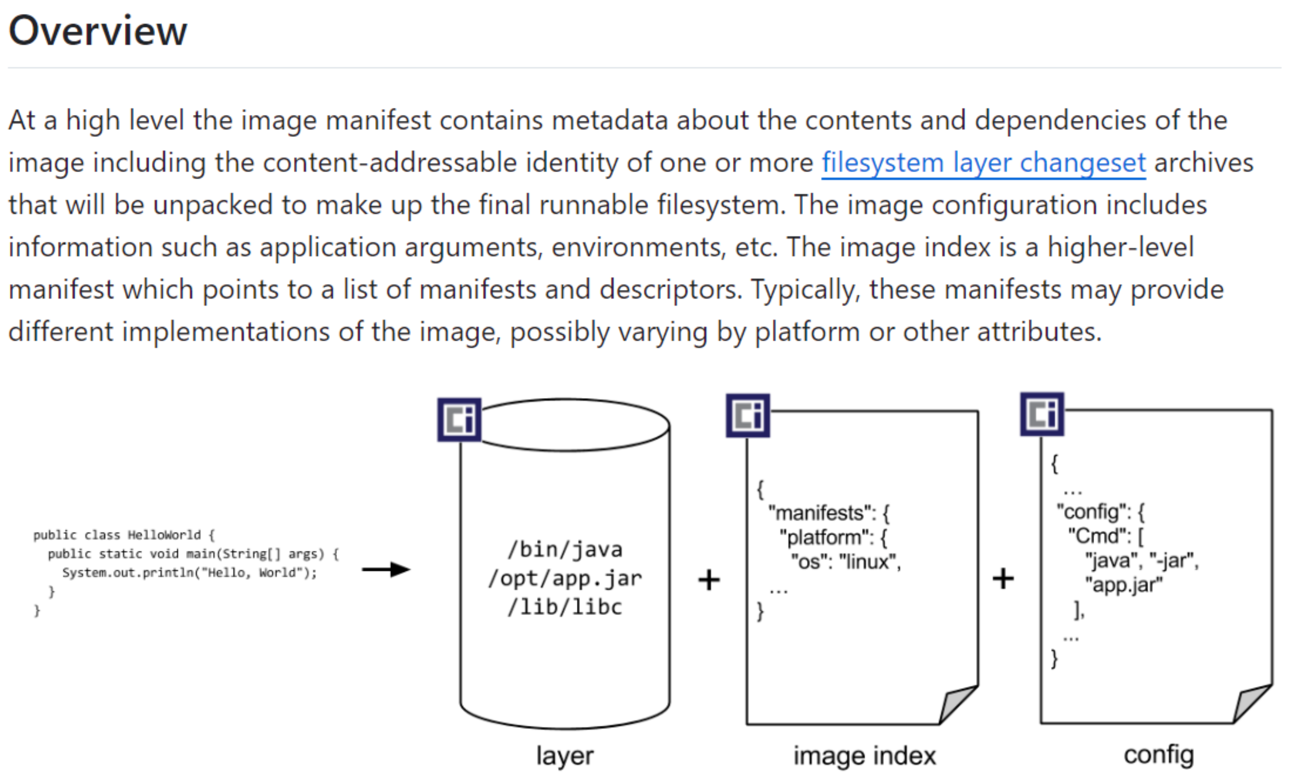
Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB Image Layers (R/O) ubuntu:15.04 Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

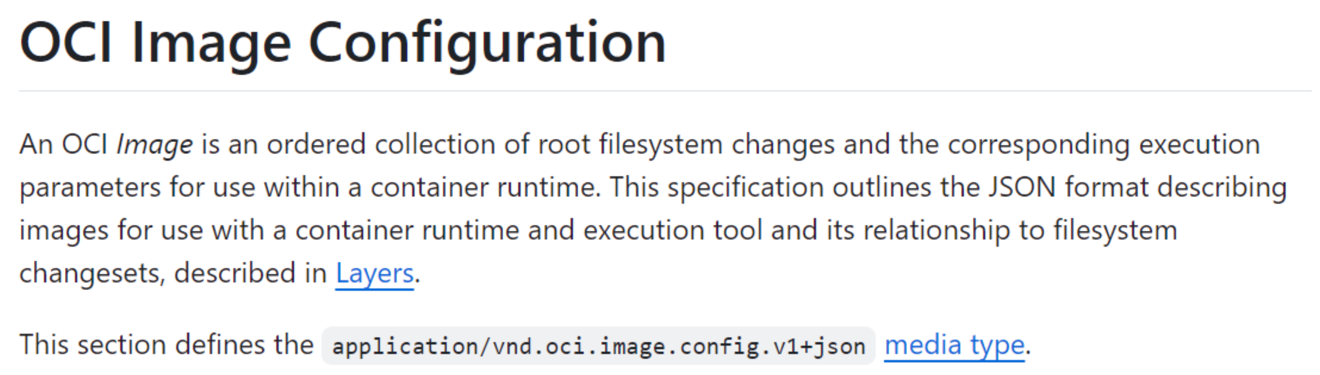
Claim 1	Accused Instrumentalities
	 Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/  Container environment The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

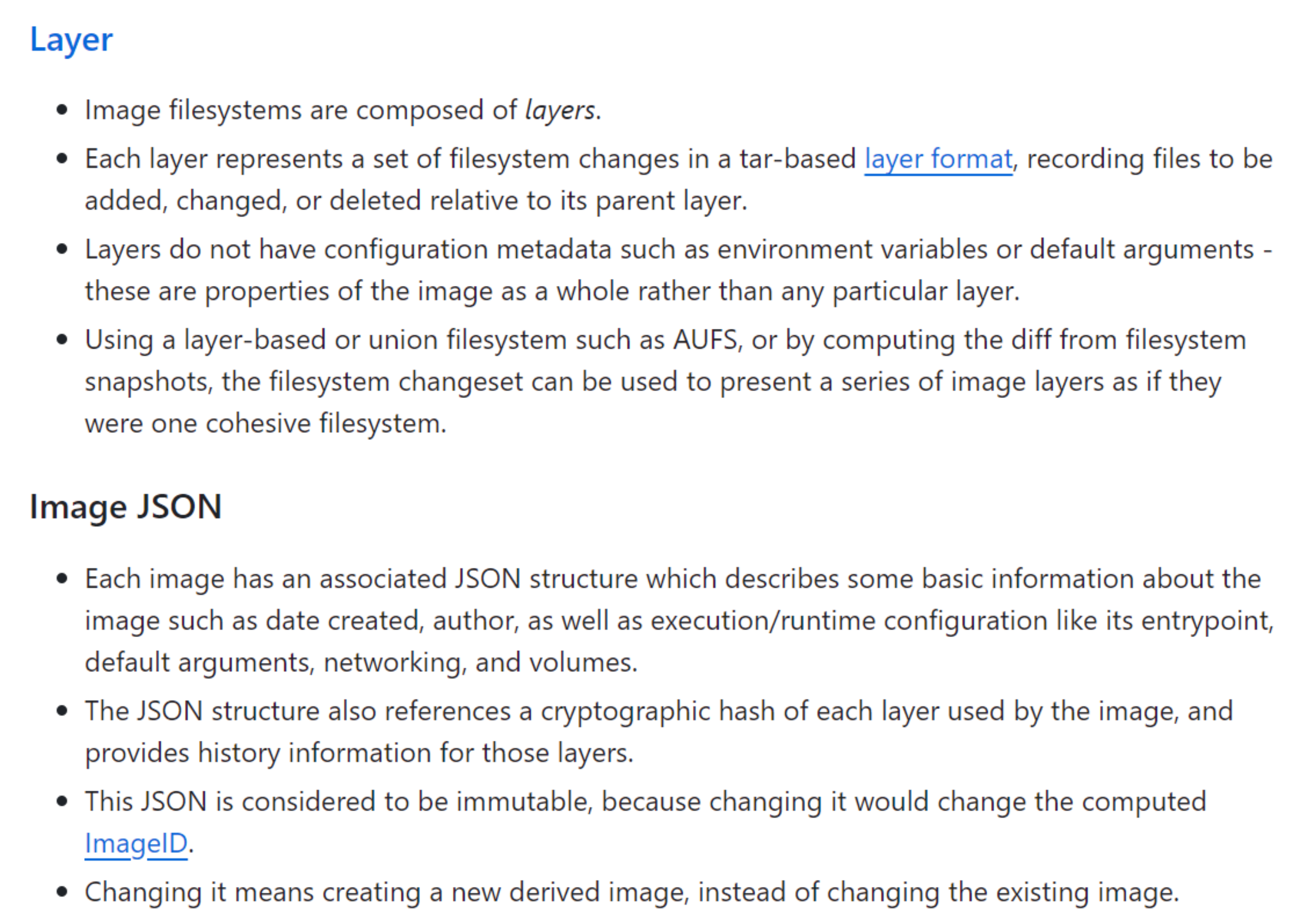
Claim 1	Accused Instrumentalities
	 **Images** A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/  **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	 Open Container Initiative Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities

	 ## Overview
	
	At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.
	
	`public class HelloWorld {`
	`  public static void main(String[] args) {`
	`    System.out.println("Hello, World");`
	`  }`
	`}`
	
	`/bin/java` `/opt/app.jar` `/lib/libc` (layer) + `{ "manifests": { "platform": { "os": "linux", ... }` (image index) + `{ ... "config": { "Cmd": [ "java", "-jar", "app.jar" ], ... }` (config)
	
	https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	 **OCI Image Configuration** An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

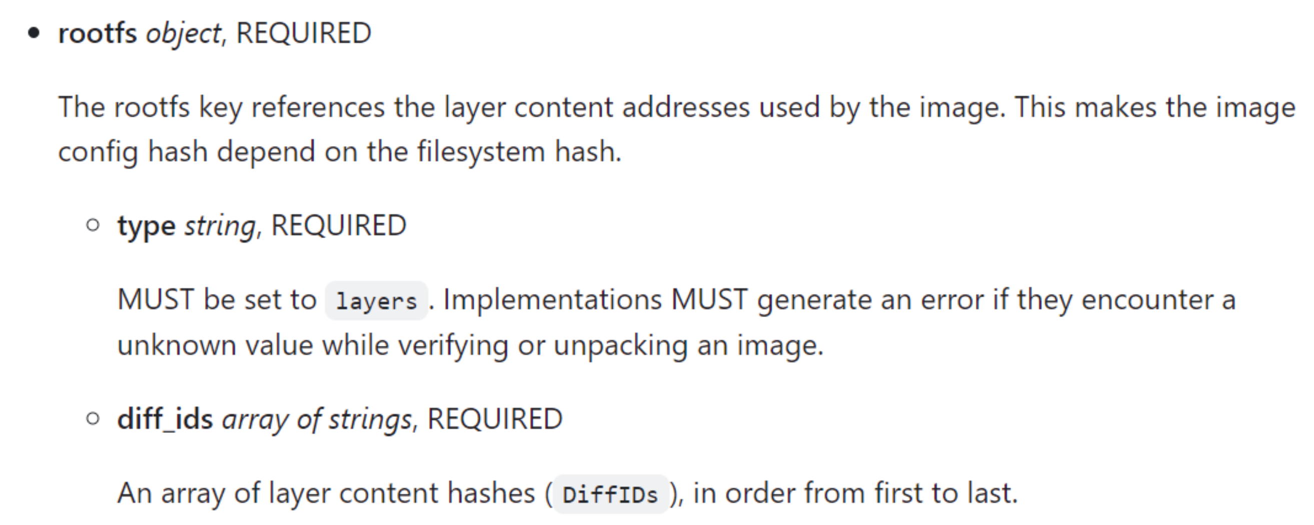
Claim 1	Accused Instrumentalities
	 https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Layer

- Image filesystems are composed of *layers*.
- Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer.
- Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer.
- Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem.

Image JSON

- Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes.
- The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.
- This JSON is considered to be immutable, because changing it would change the computed ImageID.
- Changing it means creating a new derived image, instead of changing the existing image.

Claim 1	Accused Instrumentalities
	 • **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ◦ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ◦ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md